Improving Query Optimization for Disjunctive Datalog*

Chiara Cumbo[1], Wolfgang Faber[2], and Gianluigi Greco[3]

[1] Dipartimento di Matematica, Università della Calabria, 87030 Rende, Italy
cumbo@mat.unical.it
[2] Institut für Informationssysteme, TU Wien, 1040 Wien, Austria
faber@kr.tuwien.ac.at
[3] DEIS, Università della Calabria, 87030 Rende, Italy
ggreco@si.deis.unical.it

Abstract. In this paper we present a technique for the optimization of (partially) bound queries over disjunctive deductive databases. In particular, we extend the magic-set optimization technique (originally defined for non-disjunctive deductive databases) to the disjunctive case. The method presented in this paper improves a similar approach presented in [7] by reducing the number of additionally introduced predicates and rules.
One drawback, which is intrinsic to both techniques, is that redundant rules may be created frequently. In order to overcome this, we present a method for identifying such superfluous rules, which may subsequently be deleted without changing the semantics, thereby reducing the size of the rule-base, an important performance factor.

1 Introduction

There is a growing body of work on answering queries over disjunctive deductive databases [5, 8] that have been proposed in the literature. In particular, the efficient evaluation of queries in disjunctive deductive databases and the definition of efficient evaluation algorithms for assigning semantics to disjunctive databases have been the subject of several proposals. Most of these proposals are based on the definition of efficient fixpoint algorithms and optimization is achieved by using heuristics.

The first attempt to use a different optimization technique for the evaluation of bound disjunctive Datalog queries was introduced in [6]. This approach stems from the observation that bottom-up query answering methods tend to explore a much larger search space than what is strictly necessary, whereas top-down processing uses the information provided by the query to prune the search space, thus performing a more focused search, by extending binding propagation methods defined for Datalog queries to disjunctive Datalog queries.

* This work was supported by the EC under projects IST-2002-33570 INFOMIX and IST-2001-37004 WASP.

Its main idea is to adapt the well-known magic-set method for the case of disjunctive programs. The magic-set method is a strategy for simulating the top-down evaluation of a query by modifying the original program by means of certain rules, acting as filters for the relevant information needed for the query. It essentially consists of three separate steps: (1) An *Adornment step* in which the relationship between a bound argument in the rule head and the bindings in the rule body is made explicit. (2) A *Generation step* in which the adorned program is used to generate the *magic rules* which simulate the top-down evaluation scheme. (3) A *Modification step* in which the adorned rules are modified by the magic rules generated in step (2); these rules are called *modified rules.*

An elaborated extension of the magic-set technique to the disjunctive case has been published in [7]. In this paper we present an alternative extension, which needs fewer additional predicates and rules, by exploiting a direct coupling with the DLV system [2, 3]. Due to space limits, we will not describe the architecture of the system in detail and refer to [4].

A crucial point of both approaches is that during the rewriting redundant rules are created frequently. Since the number of rules in a program is a critical performance factor, these redundancies can deteriorate runtime behavior. In extreme cases, this overhead alone can outweigh the benefits of the optimization.

Unfortunately, it is infeasible to identify all cases of redundancies because of complexity results for the subsumption problem. However, we formulate a technique which can identify some of these cases in polynomial time. In principle, it consists of a greedy approach to detecting subsumption. Informal performance tests have shown a positive impact on a number of problems. In particular, the proposed approach yields a speedup on a number of problems and reduces overhead for other problems, while its introduced overhead is only mild.

2 Preliminaries on Disjunctive Datalog

A *disjunctive rule* (*rule*, for short) r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \texttt{ v } \cdots \texttt{ v } a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k$, not $b_{k+1}, \ldots,$ not b_m is the *body* of r. Moreover, $H(r) = \{a_1, \ldots, a_n\}$ is the set of the literals in the head and $B(r) = B^+(r) \cup B^-(r)$ is the set of the body literals, where $B^+(r)$ (the *positive body*) is $\{b_1, \ldots, b_k\}$ and $B^-(r)$ (*the negative body*) is $\{b_{k+1}, \ldots, b_m\}$. Finally, if the body is empty (i.e. $k = m = 0$), r is called a fact, and we usually omit the " :- "

sign. A predicate defined only by facts is called *EDB predicate*, while a predicate defined by the rules of the program is called *IDB predicate*).

A *disjunctive datalog program* (alternatively, *disjunctive logic program*, *disjunctive deductive database*) $\mathcal{P}$ is a finite set of rules. A Datalog$^{\vee}$ program $\mathcal{P}$ (i.e., such that $\forall r \in \mathcal{P} : B^{-}(r) = \emptyset$) is called *positive.* A Datalog$^{\vee}$ program such that each rule has exactly one head predicate is referred to as Datalog program. Note that in this paper only programs are considered, for which $\forall r \in \mathcal{P} : H(r) \neq \emptyset$ holds.

The model-theoretic semantics assigns to any program $\mathcal{P}$ the set $\mathcal{SM}(\mathcal{P})$ of its *stable models.* Let $\mathcal{P}$ be a disjunctive datalog program and $\mathcal{F}$ be a set of facts. Then, a *query* $\mathcal{Q}$ for $\mathcal{P}$ over the set of facts F is a conjunction of atoms, that defines a mapping from the facts of $\mathcal{P} \cup \mathcal{F}$ to a finite (possibly empty) set of finite (possibly empty) tuples of atoms for $\mathcal{Q}$. Given a query $\mathcal{Q}$ and an interpretation M of $\mathcal{P}$, $\vartheta(\mathcal{Q}, M)$ denotes the set of substitutions for the variables in $\mathcal{Q}$ such that $\mathcal{Q}$ is true in M. The answer to a query $\mathcal{Q}$ over the fact $\mathcal{F}$, under the *brave* semantics, denoted by $Ans_b(\mathcal{Q}, \mathcal{F})$, is the set $\cup_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{SM}(\mathcal{P} \cup \mathcal{F})$. The answer to a query $\mathcal{Q}$ over the fact $\mathcal{F}$, under the *cautious* semantics, denoted by $Ans_c(\mathcal{Q}, \mathcal{F})$, is the set $\cap_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{SM}(\mathcal{P} \cup \mathcal{F})$.

3 Magic-Set for Datalog Queries

In this section we briefly review the main steps of the magic-set method, for the case of Datalog queries. For details, we refer to [1].

Adornment Step. An *adorned program* is a program whose predicate symbols have associated a string α, defined on the alphabet $\{b, f\}$, of length equal to the arity of the predicate. A character b (resp. f) in the i-th position of the adornment associated with a predicate p means that the i-th argument of p is bound (resp. free), i.e., there (resp. not) is a way for passing some binding information from the query to it, by simulating a top-down evaluation.

The adornment consists of generating an adorned program, that can be used for deriving how the binding of the query propagates. The predicates occuring in the query are adorned by marking each constant argument with b, all others with f. This is accomplished by a function ***Adorn-Query*** which outputs a tuple $\langle adornedQuery, S\rangle$, where $adornedQuery$ is the adorned query and S is a stack of the adorned predicates.

Example 1. Consider the query `p(1)`, `q(2,X)`, `r(X,Y)`?, where `p` and `q` are IDB predicates, while `r` is an EDB predicate. Then, the function ***AdornQuery*** outputs the query $\mathtt{p}^{\mathtt{b}}(1)$, $\mathtt{q}^{\mathtt{bf}}(2,\mathtt{X})$, `r(X,Y)`, and pushes on the stack S the adorned predicates $\mathtt{p}^{\mathtt{b}}$ and $\mathtt{q}^{\mathtt{bf}}$. □

Once the stack of adorned predicates S has been created, we use it for adorning all rules of the program by tracing the propagation of the binding of the arguments of the query into the rules. In particular, given an adorned predicate $\mathtt{p}^{\alpha}$, each rule r having the literal $\mathtt{p(t)}$ in the head is adorned by exploiting a particular strategy that is known as *SIPS* (Sideways Information Passing Strategy), which simulates the data flow occurring in the top-down evaluation of the query.

In the following, we will refer to the algorithm implementing the SIPS as the ***AdornRule*** algorithm. Roughly, it takes as input a rule r to be adorned w.r.t. $\mathtt{p}^{\alpha}$, the stack S of adorned predicates to be used for propagating the binding (constructed by ***AdornQuery***), and an additional argument *adornedPredicates*, containing all the adorned predicates generated so far. Initially, after the call to ***AdornQuery***, we have that *adornedPredicates* and S are equal.

Then, iteratively, a *preferred* literal among the ones that are *admissible* in the rule, that is among the ones that can be adorned respecting the SIPS, is chosen. The iterations continue until all literals have been considered or no more *admissible* literal is left.

Thus, a natural way (the one usually described in the literature) for generating the adorned program consists of the following steps: (1) extract an adorned predicate $\mathtt{p}^{\alpha}$ from S, (2) use $\mathtt{p}^{\alpha}$ for propagating the binding into the program, and collect in S all the other adorned predicates generated, (3) if S is not empty return to Step (1).

Example 2. Continuing from Ex. 1, consider the rules $\mathtt{p(X)}$:- $\mathtt{q(X,X)}$. and $\mathtt{q(X,Y)}$:- $\mathtt{r(X,Y),q(X,Y)}$. First, $\mathtt{p}^{\mathtt{b}}$ is popped from S, generating the rule $\mathtt{p}^{\mathtt{b}}\mathtt{(X)}$:- $\mathtt{q}^{\mathtt{bb}}\mathtt{(X,X)}$. and pushing $\mathtt{q}^{\mathtt{bb}}$ onto the stack. Then, $\mathtt{q}^{\mathtt{bb}}$ is popped, generating the rule $\mathtt{q}^{\mathtt{bb}}\mathtt{(X,Y)}$:- $\mathtt{r(X,Y)},\mathtt{q}^{\mathtt{bb}}\mathtt{(X,Y)}$., and finally $\mathtt{q}^{\mathtt{bf}}$ is popped, giving rise to $\mathtt{q}^{\mathtt{bf}}\mathtt{(X,Y)}$:- $\mathtt{r(X,Y)},\mathtt{q}^{\mathtt{bb}}\mathtt{(X,Y)}$. □

Generation and Modification Steps. We use the adorned program for the generation of the magic rules. First, we define a transformation over the adorned literals. For each adorned literal $\mathtt{p}^{\alpha}$, its *magic version* consists of the literal $\mathtt{magic\_p}^{\alpha}$ in which we eliminate the variables which are free w.r.t. α. In the following, given an adorned predicate $\mathtt{p}^{\alpha}$, we often denote its magic version by $\boldsymbol{magic}(\mathtt{p}^{\alpha})$; moreover, given a conjunction of $\mathtt{n}$ adorned literals $\mathtt{p}_1^{\alpha_1},\ldots,\mathtt{p}_n^{\alpha_n}$, we denote by $\boldsymbol{magic}(\mathtt{p}_1^{\alpha_1},\ldots,\mathtt{p}_n^{\alpha_n})$ the conjunction $\boldsymbol{magic}(\mathtt{p}_1^{\alpha_1})$,..., $\boldsymbol{magic}(p_n^{\alpha_n})$. Then, for each adorned predicate $\mathtt{p}^{\alpha}$ in the body of any adorned rule r_a we generate a magic rule r_m such that (i) the head of r_m consists of its magic version, and (ii) the body of r_m consists of the magic version of the head literal of r_a, and those predicates of r_a which are relevant for propagating the binding on $\mathtt{p}^{\alpha}$.

In the following we denote by $\boldsymbol{Generate}(r_a)$ the set of magic rules, generated by the procedure outlined above.

Finally, the modification step consists in the modification of each adorned rule; specifically, for each adorned rule whose head is $\mathtt{p}^{\alpha}(\mathtt{X})$, we extend the rule body by inserting $\boldsymbol{magic}(\mathtt{p}^{\alpha})$. The final program will contain only the rules which are needed to answer the query.

Example 3. Continuing from Ex. 2, the corresponding magic rules are $\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X},\mathtt{X})$:- $\mathtt{magic\_p}^{\mathtt{b}}(\mathtt{X})$., $\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$., $\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{magic\_q}^{\mathtt{bf}}(\mathtt{X}),\mathtt{r}(\mathtt{X},\mathtt{Y})$.

The associated modified rules will be $\mathtt{p}^{\mathtt{b}}(\mathtt{X})$:- $\mathtt{magic\_p}^{\mathtt{b}}(\mathtt{X}),\mathtt{q}^{\mathtt{bb}}(\mathtt{X},\mathtt{X})$., $\mathtt{q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{magic\_q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y}),\mathtt{r}(\mathtt{X},\mathtt{Y}),\mathtt{q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$., and
$\mathtt{q}^{\mathtt{bf}}(\mathtt{X},\mathtt{Y})$:- $\mathtt{magic\_q}^{\mathtt{bf}}(\mathtt{X}),\mathtt{r}(\mathtt{X},\mathtt{Y}),\mathtt{q}^{\mathtt{bb}}(\mathtt{X},\mathtt{Y})$. □

Query Rules. In the steps described so far, the query is used only for deriving the original set of adorned predicates. In fact, for each adorned predicate of the query $\mathtt{g}_{\mathtt{i}}^{\alpha_{\mathtt{i}}}$, (i) the fact $\boldsymbol{magic}(\mathtt{g}_{\mathtt{i}}^{\alpha_{\mathtt{i}}})$ has to be asserted, (ii) a rule of the form $\mathtt{g}_{\mathtt{i}}(\mathtt{t}_{\mathtt{i}})$:- $\mathtt{g}_{\mathtt{i}}^{\alpha_{\mathtt{i}}}(\mathtt{t}_{\mathtt{i}})$ has to be introduced.

Note that Step (1) is necessary in order to provide the starting point for the top-down evaluation, while Step (2) is necessary in order to provide the answer to the original query. We assume that this is done inside a function called ***BuildQueryRules***.

Example 4. Continuing from Ex. 3, the generated query rules are $\mathtt{magic\_p}^{\mathtt{b}}(1)$., $\mathtt{magic\_q}^{\mathtt{bf}}(2)$., $\mathtt{p}(1)$:- $\mathtt{p}^{\mathtt{b}}(1)$., and $\mathtt{q}(2,\mathtt{X})$:- $\mathtt{q}^{\mathtt{bf}}(2,\mathtt{X})$. □

4 Magic-Set Method for Datalog$^{\vee}$ Programs

In this section we present a technique for the optimization of disjunctive Datalog queries, which has been implemented and integrated into the DLV system [2, 3]. One proposal for extending the magic-set technique to disjunctive Datalog queries has already been proposed in [7], and, in fact, the spirit of the present paper is close to this approach.

Yet, there are important differences; in particular, we do not use any additional predicates (called *collecting predicates* in [7]) or any additional rules for adorning disjunctive rules. Indeed, such predicates and rules are intrinsically needed in [7], as that technique consists of a rewriting into an equivalent program to be evaluated by DLV; conversely, our technique has been designed for being integrated into the core of the system, and, hence, it may use many optimization strategies resulting in a significant speed-up of the computation.

As a result of the application of these new ideas, of the great attention in the implementation issues we obtain an efficient implementation (especially if compared with the prototypes proposed in the literature), whose designing has been left as a material for further research in [7].

The main algorithm, called ***DisjunctiveMagicSet***, is reported in Figure 1. We point out that despite all the traditional implementations of the magic-set technique, our proposal is not conceptually divided into the traditional *adornment*, *generation*, and *modification* step; indeed, it exploits 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 it is used for processing each rule r of a given program $\mathcal{P}$, by contextually generating the associated modified and magic rules. Moreover, we point out that the algorithm can be used for propagating the binding into non-disjunctive rules as well.

In more detail, the algorithm takes in input a positive disjunctive datalog program $\mathcal{P}$ (without integrity constraints) and a query $\mathtt{g_1(t_1), \ldots, g_n(t_n)}$. If the query contains some IDB predicates that can be used for propagating the binding, it outputs a program $\mathcal{P}'$ consisting of a set of *modified* rules, of *magic* rules, and of *query* rules. The adorned query and the corresponding magic rules are built in the steps *2* and *3* by means of the functions ***AdornQuery*** and ***BuildQueryRules***, respectively.

Example 5. Consider the disjunctive datalog program

$$\begin{array}{l}\mathtt{p(X) \vee p(Y) :\!- a(X,Y), r(X).}\\ \mathtt{r(Y) :\!- p(Y).}\end{array}$$

and the query $\mathtt{p(1)}$? over the set of facts $\{\mathtt{a(1,2)},\ \mathtt{a(2,3)}\}$. In the algorithm of Figure 1, the function ***AdornQuery*** outputs the query $\mathtt{p^b(1)}$, and pushes on the stack S the adorned predicate $\mathtt{p}^\alpha$. Moreover, the function ***BuildQueryRules*** outputs the rule $\mathtt{p(X) :\!- p^b(X)}$ and the fact $\mathtt{magic\_p^b(1)}$. □

The central part of the algorithm consists of the steps *4-13*, that are repeated until the stack S becomes empty, i.e., until there is no further adorned predicate to be propagated. Next, we briefly explain the main ideas that are used for propagating the binding into disjunctive rules, and that are exploited in the implementation.
Let $\mathtt{p}^\alpha$ be an adorned predicate that has been removed from the stack S in step *5*, and consider a disjunctive rule r in $\mathcal{P}$ of the form

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

```
Input: A Datalog^∨ program $\mathcal{P}$, and a query $g_1(t_1), \ldots, g_n(t_n)$.
Output: The optimized program $\mathcal{P}'$.
var $S$: stack of adorned predicates; adornedQuery: set of literals;
    modifiedRules, magicRules: set of rules;
begin
 1. if $g_1(t_1), \ldots, g_n(t_n)$ has some IDB predicate then
 2.   $\langle adornedQuery, S\rangle := \mathbf{AdornQuery}(g_1(t_1), \ldots, g_n(t_n)\ ,\ \mathcal{P})$;
 3.   $\langle modifiedRules, magicRules\rangle := \mathbf{BuildQueryRules}(adornedQuery)$;
 4.   while $S \neq \emptyset$ do
 5.     $p^\alpha := S.\mathbf{pop}()$;
 6.     for each rule $r \in \mathcal{P}$: $p(t) \vee p_1(t_1) \vee \ldots \vee p_n(t_n) \text{ :- } q_1(s_1), \ldots, q_m(s_m)$ do
 7.       let $r_s$ be the rule of the form
              $p(t) \text{ :- not } p_1(t_1), \ldots, \text{not } p_n(t_n), q_1(s_1), \ldots, q_m(s_m)$;
 8.       let $r_a := \mathbf{Adorn}(r_s, p^\alpha, S)$ of the form
              $p^\alpha(t) \text{ :- not } p_1^{\alpha_1}(t_1), \ldots, \text{not } p_n^{\alpha_n}(t_n), q_1^{\beta_1}(s_1), \ldots, q_m^{\beta_m}(s_m)$;
 9.       $magicRules := magicRules \bigcup \mathbf{Generate}(r_a)$;
10.       let $r_m$ be the modified rule of the form
              $p^\alpha(t) \vee p_1^{\alpha_1}(t_1) \vee \ldots \vee p_n^{\alpha_n}(t_n) :-$
                  $\mathbf{magic}(p^\alpha(t), p_1^{\alpha_1}(t_1), \ldots, p_n^{\alpha_n}(t_n)), q_1(s_1)^{\beta_1}, \ldots, q_m^{\beta_m}(s_m)$;
11.       $modifiedRules := modifiedRules \bigcup \{r_m\}$;
12.     end for
13.   end while
14.   $P' := magicRules \cup modifiedRules$;
15. return $\mathcal{P}'$;
16. end if
end.
```

Fig. 1. Algorithm ***DisjunctiveMagicSet***

in which the binding of p^α must be propagated due to the presence of a predicate $p(t)$ in its head. The difference w.r.t. the case of normal programs is that the binding have to be propagated also in the predicates $p_1(t_1), \ldots, p_n(t_n)$ occurring in the head. A simple idea for reusing the standard procedures consists of replacing r with a rule r_s without disjunction of the form

$$r_s :\ p(t) \text{ :- not } p_1(t_1), \ldots, \text{not } p_n(t_n), q_1(s_1), \ldots, q_m(s_m).$$

that can be adorned in the usual manner, since it is not disjunctive. The reason for pushing the predicates $p_1(t_1), \ldots, p_n(t_n)$ in the body by negating them, lies in the fact that these predicates cannot be used for binding variables, as they actually occur only in the head of the original rule.

The rule r_s is then adorned by means of the function ***AdornRule*** informally described in the previous section. Here we recall that it takes in input a rule and an adorned predicate and outputs the adorned rule, while pushing on the stack S all the adorned predicates that have been generated during its execution. Hence, in step *8* we assume that the rule

r_a obtained by means of this adornment is of the form

$$r_a: \ \mathtt{p}^{\alpha}(\mathtt{t}) \ \mathtt{:-}\, \mathtt{not}\ \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1), \dots, \mathtt{not}\ \mathtt{p}_\mathtt{n}^{\alpha_\mathtt{n}}(\mathtt{t}_\mathtt{n}), \mathtt{q}_1^{\beta_1}(\mathtt{s}_1), \dots, \mathtt{q}_\mathtt{m}^{\beta_\mathtt{m}}(\mathtt{s}_\mathtt{m}).$$

where $\alpha_1, \dots, \alpha_\mathtt{n}, \beta_1, \dots \beta_\mathtt{m}$ are generic adornment strings.

Example 6. Consider again the program and the query of Example 5, and let $\mathtt{p}^\mathtt{b}$ be the adorned predicate to be processed. The only rule in which the binding can propagate is the disjunctive one $\mathtt{p(X)\ v\ p(Y)\ :-\ a(X,Y), r(X)}$. Hence, when propagating on the predicate $\mathtt{p(X)}$ we preliminarily construct the standard rule $\mathtt{p(X)\ :-\, not\ p(Y), a(X,Y), r(X)}$. whose adornment is $\mathtt{p}^\mathtt{b}\mathtt{(X)\ :-\, not\ p}^\mathtt{b}\mathtt{(Y), a(X,Y), r}^\mathtt{b}\mathtt{(X)}$. Moreover, it is important to note that the binding of $\mathtt{p}^\mathtt{b}$ will also propagate on $\mathtt{p(Y)}$ by producing the adorned rule $\mathtt{p}^\mathtt{b}\mathtt{(Y)\ :-\, not\ p}^\mathtt{b}\mathtt{(X), a(X,Y), r}^\mathtt{b}\mathtt{(X)}$. Note that in both cases the predicate $\mathtt{r}^\mathtt{b}$ is pushed onto the stack S, and this predicate will generate the adorned rule $\mathtt{r}^\mathtt{b}\mathtt{(Y)\ :-\, p}^\mathtt{b}\mathtt{(Y)}$. □

The algorithm subsequently uses the adorned rule r_a for generating the magic rules in the step *9*. As r_a is not a disjunctive rule, this is essentially the same for the technique in the case of Datalog queries, and, hence, is carried out by means of the function ***Generate*** sketched in the previous section.

Example 7. In the program of Example 6 the propagation of the predicate $\mathtt{p}^\mathtt{b}$ generates the following magic rules

$$\begin{array}{l}
\mathtt{magic\_r}^\mathtt{b}\mathtt{(X)\ :-\, magic\_p}^\mathtt{b}\mathtt{(X), p}^\mathtt{b}\mathtt{(Y), a(X,Y)}.\\
\mathtt{magic\_p}^\mathtt{b}\mathtt{(Y)\ :-\, magic\_p}^\mathtt{b}\mathtt{(X), r}^\mathtt{b}\mathtt{(X), a(X,Y)}.\\
\\
\mathtt{magic\_r}^\mathtt{b}\mathtt{(X)\ :-\, magic\_p}^\mathtt{b}\mathtt{(Y), p}^\mathtt{b}\mathtt{(X), a(X,Y)}.\\
\mathtt{magic\_p}^\mathtt{b}\mathtt{(X)\ :-\, magic\_p}^\mathtt{b}\mathtt{(Y), r}^\mathtt{b}\mathtt{(X), a(X,Y)}.
\end{array}$$

where the first group is generated from $\mathtt{p}^\mathtt{b}\mathtt{(X)\ :-\, not\ p}^\mathtt{b}\mathtt{(Y), a(X,Y), r}^\mathtt{b}\mathtt{(X)}$, while the second one from $\mathtt{p}^\mathtt{b}\mathtt{(Y)\ :-\, not\ p}^\mathtt{b}\mathtt{(X), a(X,Y), r}^\mathtt{b}\mathtt{(X)}$. Moreover, by considering also the rule $\mathtt{r}^\mathtt{b}\mathtt{(Y)\ :-\, p}^\mathtt{b}\mathtt{(Y)}$ we derive $\mathtt{magic\_p}^\mathtt{b}\mathtt{(Y)\ :-\, magic\_r}^\mathtt{b}\mathtt{(Y)}$. □

Finally, step *10* is devoted to the generation of the modified rule. The problem is that we need to reconstruct a disjunctive rule by pushing back each adorned predicate $\mathtt{p_i}$ into the head, that has been translated into the body in the *adornment* step. Hence, we construct the rule r_m of the form

$$\begin{array}{ll}
\mathtt{p}^{\alpha}(\mathtt{t})\ \mathtt{v}\ \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1)\ \mathtt{v}\ \dots\ \mathtt{v}\ \mathtt{p}_\mathtt{n}^{\alpha_\mathtt{n}}(\mathtt{t}_\mathtt{n}) :- & \boldsymbol{magic}(\mathtt{p}^{\alpha}(\mathtt{t}), \mathtt{p}_1^{\alpha_1}(\mathtt{t}_1), \dots, \mathtt{p}_\mathtt{n}^{\alpha_\mathtt{n}}(\mathtt{t}_\mathtt{n})),\\
& \mathtt{q}_1(\mathtt{s}_1)^{\beta_1}, \dots, \mathtt{q}_\mathtt{m}^{\beta_\mathtt{m}}(\mathtt{s}_\mathtt{m}).
\end{array}$$

and outputs a conjunction of magic predicates associated to the ones in input; for instance given $\mathtt{p}^{\mathtt{bfb}}(\mathtt{X},\mathtt{Y},\mathtt{Z})$ it outputs $\mathtt{magic\_p}^{\mathtt{bfb}}(\mathtt{X},\mathtt{Z})$

Example 8. By means of the application of the generation step to our running example, we derive the following set of modified rules

$$\begin{array}{l}\mathtt{p}^{\mathtt{b}}(\mathtt{X})\ \mathtt{v}\ \mathtt{p}^{\mathtt{b}}(\mathtt{Y}) \text{ :- } \mathtt{magic\_p}^{\mathtt{b}}(\mathtt{X}), \mathtt{magic\_p}^{\mathtt{b}}(\mathtt{Y}), \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{r}^{\mathtt{b}}(\mathtt{X}).\\ \mathtt{p}^{\mathtt{b}}(\mathtt{Y})\ \mathtt{v}\ \mathtt{p}^{\mathtt{b}}(\mathtt{X}) \text{ :- } \mathtt{magic\_p}^{\mathtt{b}}(\mathtt{Y}), \mathtt{magic\_p}^{\mathtt{b}}(\mathtt{X}), \mathtt{a}(\mathtt{X},\mathtt{Y}), \mathtt{r}^{\mathtt{b}}(\mathtt{X}).\end{array}$$

Finally, for the sake of completeness we point out that the rewritten program $\mathcal{P}'$ is constituted by the above rules, plus the magic rules of Example 7 and the following rule $\mathtt{r}^{\mathtt{b}}(\mathtt{X}) \text{ :- } \mathtt{magic\_r}^{\mathtt{b}}(\mathtt{X}), \mathtt{p}^{\mathtt{b}}(\mathtt{X}).$ obtained by considering the propagation of $\mathtt{r}^{\mathtt{b}}$. □

As pointed out before, the spirit of this approach is the same of the one proposed in [7] (even though further optimized), and, hence, the soundness and completeness of the above algorithm follow easily.

Proposition 1. *Let $\mathcal{P}$ be a Datalog$^{\vee}$ program, let $\mathcal{Q}$ be a query, and let $\mathcal{P}'$ be the result of the Algorithm **DisjunctiveMagicSet**. Then, $\mathcal{P} \equiv_{\mathcal{Q}} \mathcal{P}'$ under both the cautious and brave semantics.*

5 Identifying Redundant Rules

In this section we address one drawback of the magic-set method when extended to disjunctive Datalog programs, consisting of the generation of a number of redundant, useless rules, which may reduce the optimization effect. For a better understanding of this point, let us return to Example 8; as the careful reader may have noticed, the first two modified rules coincide, even though they are generated by adorning different head atoms of the same rule. Such a redundancy is, indeed, intrinsic in any possible implementation; nonetheless, we point out that compared to [7], our rewriting reduces drastically the presence of redundant rules, since it does not require any additional predicates or any additional rules, which are an additional source of redundancy.

In our implementation, great attention has been devoted to this issue, since we have experimentally observed that it is crucial for the whole optimization process for several problem domains. In particular, we propose an efficient (polynomial time) heuristic for the individuation of such rules, whose relevance is not restricted to the magic-set method in itself.

In fact, this heuristic has been integrated into the core of the DLV system, in order to identify redundancy in any type of program. However, redundant rules are less frequently found in user-specified programs. We point out that the complexity results obtained for subsumption problems implies that such an heuristic cannot be complete. Still, we are able to show its soundness, i.e., all the rules identified by the heuristic are indeed subsumed by some other rule.

Before presenting the algorithm we introduce some concepts and notations. Let C be a set of literals, and $X_1, \ldots, X_n$ the variables occurring in C. Then, a *substitution* for C is a finite set of pairs $\vartheta = \{X_1/t_1, \ldots, X_n/t_n\}$ where $t_1, \ldots, t_n$ are terms (either variables or constants). Moreover, we denote by $\vartheta(C)$ the set of literals obtained from C by simultaneously replacing each occurrence of X_i by t_i $(1 \leq i \leq n)$.

Definition 1. 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 $H(r_1) \cup B(r_1)$, such that $\vartheta(H(r_1)) \subseteq H(r_2)$ and $\vartheta(B(r_1)) \subseteq B(r_2)$. A rule r_1 is *redundant* if there exists a rule r_2 such that $r_1 \sqsubseteq r_2$. $\Box$

Definition 2. Let L_1 and L_2 be two sets of literals. Then, a *matcher* for L_1 w.r.t. L_2 is a substitution ϑ for L_1 such that $\vartheta(L_1) = L_2$. $\Box$

Given two rules r_1 and r_2 of a logic program, the concept of subsumption can be restated in terms of existence of matchers.

Definition 3. Let r_1 and r_2 be two disjunctive rules, l be a literal of r_1. Then, l is a *candidate for the subsumption verification* for r_1 w.r.t. r_2 if there exists a substitution ϑ such that (i) if $l \in B(r_1)$, then there exists $l' \in B(r_2)$ such that $\vartheta(l) = \{l'\}$, and (ii) if $l \in H(r_1)$, then there exists $l' \in H(r_2)$ such that $\vartheta(l) = \{l'\}$. Moreover, ϑ is called *matcher for the candidate l.* $\Box$

Note that in the above definition, each matcher ϑ satisfying conditions 1 and 2 gives rise to at least one literal of r_2 that can be matched with l; indeed, we denote by $\mathit{matchable}(l, r_1, r_2)$ the set of all such literals of r_2.

We point out that given a literal l, three different representative situations may occur: i) there exists exactly one candidate literal, i.e., $|\mathit{matchable}(l, r_1, r_2)| = 1$, (in such a case the literal is said *deterministic*), ii) $|\mathit{matchable}(l, r_1, r_2)| = 0$, or iii) $|\mathit{matchable}(l, r_1, r_2)| > 1$, i.e., the selection is non-deterministic. Obviously, the source of the complexity of the problem lies in literals of the latter type. Note that in the point ii) above l cannot be a candidate.

In the following, given a rule r, and a set of literals L of r, we denote by $r - L$ the rule obtained by removing from r each literal in L. Moreover, given a substitution ϑ we denote by $\vartheta(r)$ the rule obtained by the application of ϑ to both the head and the body of r. Our heuristic is based on the following observation.

Proposition 2. *Let r_1 and r_2 be two disjunctive rules. Then, r_1 is subsumed by r_2 if and only if there exists an ordering of all the literals 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*

1. *each l_i is a candidate for the subsumption verification for $\vartheta(r_1 - \{l_1, \ldots, l_{i-1}\})$ w.r.t. r_2, where $\vartheta = \vartheta_1 \cup \ldots \cup \vartheta_{i-1}$, and*
2. *ϑ_i is a matcher for the candidate l_i.*

Essentially, our heuristic tries to construct the sequence $l_1, \ldots, l_m$ of the above proposition and the associated substitutions $\vartheta_1, \ldots, \vartheta_n$ in an incremental way. At the first step we choose l'_1 in r_1 and l''_1 in r_2 such that there exists a matcher ϑ_1 for l'_1 and l''_2. Then, at each successive step, say i, we choose a literal l'_i of r_1 that is a candidate for the subsumption verification. Moreover, in case there is more than one candidate, we select the one which is *preferred* according to the following definition.

Definition 4. Let r_1 and r_2 be two disjunctive rules, let $l_1, \ldots, l_j$ be a sequence of distinct literals of r_1 and $\vartheta_1, \ldots, \vartheta_j$ be a sequence of matchers. Then, a candidate literal l of $r_1 - \{l_1, \ldots, l_j\}$ is *preferable* to a candidate literal l' of $r_1 - \{l_1, \ldots, l_j\}$ w.r.t. $l_1, \ldots, l_j$ if i) $|matchable(l, r_1, r_2)| < |matchable(l', r_1, r_2)|$ or ii) $|matchable(l, r_1, r_2)| = |matchable(l', r_1, r_2)|$ and the number of distinct variables not yet matched w.r.t. $\vartheta_1, \ldots, \vartheta_j$ of l is greater than that of l'. Moreover, a literal l in r_1 is *preferred* if it does not exists a literal preferable to it. □

Note that a deterministic literal is always preferable to a non-deterministic one. Moreover, we point out that, in general, deciding whether l is a preferred literal can be done in polynomial time.

The heuristic is based on the idea of selecting the preferred literal as described in the above definition. Moreover, each time a literal l is selected, it is deleted from the rule, while other than storing the sequence $\vartheta_1, \ldots, \vartheta_m$, we simply update a unique substitution ϑ. The soundness of this approach can be easily proved.

It is worth noting that the algorithm runs in polynomial time, while the problem is NP-hard, Hence completeness cannot hold; nevertheless, several experimental results confirmed its effectiveness.

6 Conclusions

We have presented a new technique for the optimization of (partially) bound queries, that extends the magic-set method to the case of disjunctive databases. As one intrinsic drawback of this method is the generation of many redundant rules, we have also introduced a strategy for their efficient identification. We point out that the proposed heuristic can be applied also for identifying redundant rules in programs in which no optimization is performed, albeit this situation is likely to occur less frequently. We are currently working on the definition of more focused prevention strategies that can be directly integrated into the optimization module of the DLV system.

Finally, our long-term objective is to extend the application of the magic set method to the case of disjunctive programs with constraints, and, more ambitiously, to the case of general unstratified programs.

References

1. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. A Deductive System for Nonmonotonic Reasoning. In Jürgen Dix and Ulrich Furbach and Anil Nerode, editor, *Proceedings of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, number 1265 in Lecture Notes in AI (LNAI), pages 363–374, Berlin, 1997. Springer.
3. Thomas Eiter, Nicola Leone, Cristinel Mateis, Gerald Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.
4. Wolfgang Faber and Gerald Pfeifer. `DLV` homepage, since 1996. `http://www.dlvsystem.com/`.
5. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
6. Sergio Greco. Optimization of Disjunction Queries. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 441–455, Las Cruces, New Mexico, USA, November 1999. The MIT Press.
7. Sergio Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):368–385, March/April 2003. Extended Abstract appeared as [6].
8. Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.