Advanced BackJumping Techniques for Rule Instantiations

S. Perri[1] and F. Scarcello[2]

[1] Department of Mathematics, University of Calabria
I-87030 Rende (CS), Italy
perri@mat.unical.it
[2] D.E.I.S., University of Calabria
I-87030 Rende (CS), Italy
scarcello@deis.unical.it

Abstract. The interest in the area of non-monotonic reasoning and declarative logic programming is growing rapidly after the recent development of a number of Answer Set Programming (ASP) systems. The logic-based languages supported by such systems are rich enough to represent in a natural and declarative way a large number of problems from different domains. Nevertheless, the computation of the answer sets is always performed by these systems on simple ground (i.e., variable free) programs, first computed by a pre-processing phase, called instantiation. This phase may be computationally expensive, and in fact it has been recognized to be a key issue for solving real-world problems by using Answer Set Programming. Given any program P, a good instantiation for it is a ground program P' having the same answer sets as P and such that: P' can be computed efficiently from P, and P' does not contain useless rules and thus can be evaluated efficiently.
In this paper, we present a structure-based backjumping algorithm that meets the above requirements. In particular, given a rule r to be grounded, our algorithm exploits both the semantical and the structural information about r for computing efficiently all and only the ground instances of r that should be included in any sound instantiation of P. That is, from each general rule r, we are able to compute only the relevant subset of all its possible ground instances.
We have implemented this algorithm in the ASP system **DLV**, and we have carried out an experimentation activity on a collection of benchmark problems. The results are very positive, as the new technique improves sensibly the efficiency of the **DLV** system on many kind of programs.

1 Introduction

After many years of theoretical research and some years of considerable efforts on developing effective implementations, there are nowadays a number of systems that support a fully declarative programming style, called *Answer Set Programming* (ASP) [1, 2, 4, 5, 7, 6, 10, 11, 9, 12, 13, 18, 19, 22–24].

The knowledge representation language of ASP is very expressive: function-free logic programs where nonmonotonic negation may occur in the bodies of the rules, and possibly (i.e., for some systems) with classical negation and disjunction in the heads of the rules. The semantics of an ASP program P is given by its *answer sets* [15], which are subset-minimal models of P, and are "grounded" in a precise sense. The idea of answer set programming is to represent a given computational problem by an ASP program whose answer sets correspond to solutions, and then use an answer set solver to find such a solution [17].

The logic-based languages supported by these systems are quite rich and offer a wide range of syntactical constructs, in order to encode problems from different domains in a very natural and declarative way. Nevertheless, the kernel modules of all the available ASP systems operate on simple ground (i.e., variable-free) programs. Indeed, any given program P first undergoes the so called instantiation process, that computes from P a semantically equivalent ground program P'.

Since this preprocessing phase may be computationally very expensive, having a good instantiation procedure is a key feature of ASP systems. Such a procedure, also called instantiator, should be able to produce a ground program P' with the same answer sets as P such that both: P' can be computed efficiently from P, and P' does not contain useless rules, and thus can be evaluated efficiently by an ASP solver.

The main reason of large grounding programs even for small input programs is that each atom of a rule in a program $\mathcal{P}$ may be instantiated to many atoms in its Herbrand base, which leads to combinatorial explosion. However, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules. The idea is that the instantiator should generate ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$.

To this end, e.g., the **DLV** instantiator exploits the dependencies among predicates. The instantiation starts by evaluating first the rules defining predicates P_0 that depend on no other predicates (that is, only defined by facts), then the predicates P_1 that only depend on predicates in P_0, and so on. It is worthwhile noting that, if the input program is normal (i.e., $\vee$-free) and stratified, this instantiator evaluates completely the program and no further module is employed after the grounding; the program has a single answer set, namely the set of the facts and the atoms derived by the instantiation procedure. If the input program is disjunctive or unstratified, the instantiation procedure cannot evaluate completely the program.

Even in this case, at each step of the instantiation process, we have a number of predicates, that we call *solved*, such that the truth values of all their ground instances are already determined by the instantiator. For instance, all predicates in P_0 are solved, as well as all predicates that only depend on solved predicates. It follows that none of these predicates should occur in the rules (but the facts) of the ground program P' produced by the instantiator. All the predicates occurring in the rules of P' should be unsolved, and will be evaluated by the answer set solver.

Example 1. Consider the following rule

$$r_1 : a(X, Z) :- q_1(X, Z, Y), q_2(W, T, S), q_3(V, T, H), q_4(Z, H), q_5(T, S, V).$$

Suppose we know that predicates q_3, q_4, and q_5 are solved, and consider the following ground instances for r_1:

$$a(x_1, z_1) :- q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1), q_3(v_1, t_1, h_1), q_4(z_1, h_1), q_5(t_1, s_1, v_1).$$
$$a(x_1, z_1) :- q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1), q_3(v_2, t_1, h_1), q_4(z_1, h_1), q_5(t_1, s_1, v_2).$$
$$\vdots$$
$$a(x_1, y_1, z_1) :- q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1), q_3(v_{100}, t_1, h_{100}), q_4(z_1, h_{100}), q_5(t_1, s_1, v_{100}).$$

Now, assume that all these instances are applicable, that is, all instances of the atoms over solved predicates are true, and all instances over unsolved predicates could be true (i.e., they are not provably false, at this point). Then, it is easy to see that all these 10000 rules are semantically equivalent to the single instance

$$a(x_1, z_1) :- q_1(x_1, z_1, y_1), q_2(w_1, t_1, s_1).$$

Thus, we only need all the (applicable) instantiations of unsolved predicates, while the solved ones are just used to validate such instances. More precisely, we are not interested in finding all the "consistent" substitutions for all variables, but rather their restrictions to the only variables that occur in literals over unsolved predicates. We call such variables the *relevant variables* of a rule r, and any applicable ground instance projected onto the unsolved predicates (as the one shown in the above example) a *relevant instance* for r. These ground rules should be included in any sound instantiation of P.

In this paper we present a new kind of structure-based backjumping algorithm that is able to compute efficiently all and only the relevant instances of a rule r. To this end, it exploits both the semantical information on the relevant variables of r, and the structural information on how the literals in the body of r are connected each other through the variables they have in common.

It is worthwhile noting that, unlike other approaches, we do not filter-out useless instances after their computation. Rather, we directly avoid their generation.

We have implemented this algorithm in the ASP system **DLV**, and we have carried out an experimentation activity on a collection of benchmark problems. The results are very positive, as the new technique improves sensibly the efficiency of the **DLV** system on many kind of programs.

The rest of this paper is structured as follows. In Section 2, we give some basic notions of disjunctive logic programming. In Section 3, we describe how logic programs are instantiated by the **DLV** system. In Section 4, we present the algorithm *BJ_Instantiate*, that given a rule r and the set of relevant variables of r, returns a set of substitutions for these variables that are in a one-to-one correspondence with all and only the relevant (ground) instances of r. Finally, in Section 5, we describe our implementation of Algorithm *BJ_Instantiate* and discuss the results of our experiments with this algorithm.

2 Disjunctive Logic Programming

In this section, we provide a formal definition of the syntax and semantics of disjunctive logic programs.

2.1 Syntax

A variable or a constant is a *term*. An *atom* is $a(t_1, ..., t_n)$, where a is a *predicate* of arity n and $t_1, ..., t_n$ are terms. A *literal* is either a *positive literal* p or a *negative literal* `not` p, where p is an atom.[1] A *(disjunctive) rule* r has the following form:

$$a_1 \vee \cdots \vee a_n :\!- b_1, \cdots, b_k, \texttt{not } b_{k+1}, \cdots, \texttt{not } b_m, \qquad n \geq 1,\ m \geq k \geq 0$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, ..., b_k, \texttt{not } b_{k+1}, ..., \texttt{not } b_m$ is the *body* of r.

We denote by $H(r)$ the set $\{a_1, ..., a_n\}$ of the head atoms, and by $B(r)$ the set $\{b_1, ..., b_k, \neg b_{k+1}, ..., \neg b_m\}$ of the body literals. $B^+(r)$ (resp., $B^-(r)$) denotes the set of atoms occurring positively (resp., negatively) in $B(r)$. For a literal L, $var(L)$ denotes the set of variables occurring in L. For a conjunction (or a set) of literals C, $var(C)$ denotes the set of variables occurring in the literals in C, and, for a rule r, $var(r) = var(H(r)) \cup var(B(r))$. A Rule r is *safe* if each variable appearing in r appears also in some positive body literal of r, i.e., $var(r) = var(B^+(r))$.

[1] Without loss of generality, in this paper we do not consider strong negation, which is irrelevant for the instantiation process; the symbol '`not`' denotes default negation here.

An *ASP program (or disjunctive database, DDB)* $\mathcal{P}$ is a finite set of safe rules. A `not` -free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variables appear in it.

A predicate occurring only in *facts* (rules of the form $a :-$), is referred to as an *EDB* predicate, all others as *IDB* predicates. The set of facts in which *EDB* predicates occur, is called *Extensional Database (EDB)*, the set of all other rules is the *Intensional Database (IDB)*.

Please note that we make frequent use of rules without a head $:-l_1, \ldots, l_n$, called *constraints*, which are a shorthand for $false :-l_1, \ldots, l_n$, and it is also assumed that a rule $bad :- false, \texttt{not}\ bad$ is in the DDB, where $false$ and *bad* are special symbols appearing nowhere else in the DDB. So, intuitively, the body of a constraint must not be true in any answer set.

2.2 Semantics

Let $\mathcal{P}$ be a program. The *Herbrand Universe* and the *Herbrand Base* of $\mathcal{P}$ are defined in the standard way and denoted by $U_\mathcal{P}$ and $B_\mathcal{P}$, respectively.

Given a rule r occurring in a DDB, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where $\sigma : var(r) \mapsto U_\mathcal{P}$ is a substitution mapping the variables occurring in r to constants in $U_\mathcal{P}$. We denote by $ground(\mathcal{P})$ the set of all the ground instances of the rules occurring in $\mathcal{P}$.

An *interpretation* for $\mathcal{P}$ is a set of ground atoms, that is, an interpretation is a subset I of $B_\mathcal{P}$. A ground positive literal A is *true* (resp., *false*) w.r.t. I if $A \in I$ (resp., $A \notin I$). A ground negative literal `not` A is *true* w.r.t. I if A is false w.r.t. I; otherwise `not` A is false w.r.t. I.

Let r be a ground rule in $ground(\mathcal{P})$. The head of r is *true* w.r.t. I if $H(r) \cap I \neq \emptyset$. The body of r is *true* w.r.t. I if all body literals of r are true w.r.t. I (i.e., $B^+(r) \subseteq I$ and $B^-(r) \cap I \neq \emptyset$) and is *false* w.r.t. I otherwise. The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A *model* for $\mathcal{P}$ is an interpretation M for $\mathcal{P}$ such that every rule $r \in ground(\mathcal{P})$ is true w.r.t. M. A model M for $\mathcal{P}$ is *minimal* if no model N for $\mathcal{P}$ exists such that N is a proper subset of M. The set of all minimal models for $\mathcal{P}$ is denoted by $\mathrm{MM}(\mathcal{P})$.

Given a program $\mathcal{P}$ and an interpretation I, the *Gelfond-Lifschitz (GL) transformation* of $\mathcal{P}$ w.r.t. I, denoted $\mathcal{P}^I$, is the set of positive rules

$$\mathcal{P}^I = \{\ a_1 \vee \cdots \vee a_n :- b_1, \cdots, b_k \mid a_1 \vee \cdots \vee a_n :- b_1, \cdots, b_k, \not b_{k+1}, \cdots, \not b_m \text{ is in ground}(\mathcal{P}) \text{ and } b_i \notin I, \text{ for all } k < i \leq m\}$$

Definition 1. [20, 15] Let I be an interpretation for a program $\mathcal{P}$. I is an *answer set* for $\mathcal{P}$ if $I \in \mathrm{MM}(\mathcal{P}^I)$ (i.e., I is a minimal model for the positive program $\mathcal{P}^I$). □

3 Instantiation of Disjunctive Logic Programs: DLV's Strategy

In this section, we provide a short description of the overall instantiation module of the **DLV** system, and focus on the "heart" procedure of this module which produces the ground instances of a given rule.

An input program $\mathcal{P}$ is first analyzed from the parser, which also builds the extensional database from the facts in the program, and encodes the rules in the intensional database in a suitable way. Then, a rewriting procedure (see [14]), optimizes the rules in order to get an equivalent program $\mathcal{P}'$ that can be instantiated more efficiently and that can lead to a smaller ground program. At this point, another module of the instantiator computes the dependency graph of $\mathcal{P}'$, its connected components, and a

topological ordering of these components. Finally, $\mathcal{P}'$ is instantiated one component at a time, starting from the lowest components in the topological ordering, i.e., those components that depend on no other component, according to the dependency graph.

3.1 General Instantiation Algorithm

The aim of the instantiator is mainly twofold: (i) to evaluate ($\vee$-free) stratified program components, and (ii) to generate the instantiation of disjunctive or unstratified components (if the input program is disjunctive or unstratified).

In order to evaluate efficiently stratified programs (components), **DLV** uses an improved version of the generalized semi-naive technique [26] implemented for the evaluation of linear and non-linear recursive rules.

If the input program is normal (i.e., $\vee$-free) and stratified, the instantiator evaluates completely the program and no further module is employed after the grounding; the program has a single answer set, namely the set of the facts and the atoms derived by the instantiation procedure. If the input program is disjunctive or unstratified, the instantiation procedure cannot evaluate completely the program. However, the optimization techniques mentioned above are useful to generate efficiently the instantiation of the non-monotonic part of the program. Two aspects are crucial for the instantiation:

(a) the number of generated ground rules,
(b) the time needed to generate such an instantiation.

The size of the generated instantiation is important because it strongly influences the computation time of the other modules of the system. A slower instantiation procedure generating a smaller grounding may be preferable to a faster one generating a large grounding. However, the time needed by the former can not be ignored otherwise we could not really have a computation time gain.

The main reason of large groundings even for small input programs is that each atom of a rule in $\mathcal{P}$ may be instantiated to many atoms in $B_{\mathcal{P}}$, which leads to combinatorial explosion. However, most of these atoms may not be derivable whatsoever, and hence such instantiations do not render applicable rules. The instantiator module generates ground instances of rules containing only atoms which can possibly be derived from $\mathcal{P}$.

3.2 Rule Instantiation

In this section, we describe the process of rule instantiation – the "heart" of the instantiation module – as it is currently implemented in **DLV**.

The algorithm *Instantiate*, shown in Figure 1, generates all the possible instantiations for a rule r of a program $\mathcal{P}$. When this procedure is called, for each predicate p occurring in the body of r we are given its extension, as a set I_p containing all its ground instances. We say that the mapping $\theta : var(r) \rightarrow U_{\mathcal{P}}$ is a valid substitution for r if it is valid for every literal occurring in its body, i.e., if for every positive literal p (resp., negative literal $\neg p$) in $B(r)$, $\theta p \in I_p$ (resp. $\theta p \notin I_p$) holds. *Instantiate* outputs all such valid substitutions for r, which are in a one-to-one correspondence with the set of all possible ground instances of r.

Note that, since the rule is safe, each variable occurring either in a negative literal or in the head of the rule appears also in some positive body literal. For the sake of presentation, we assume that the body is ordered in a way such that any negative literal always follows the positive atoms containing its variables. Actually, **DLV** has a specialized module that computes a clever ordering of the body (e.g., exploiting

```
Algorithm Instantiate
Input  R: Rule, I: Set of instances for the predicates occurring in B(R);
Output S: Set of Total Substitutions;
var  L: Literal, B: List of Atoms, θ: Substitution, MatchFound: Boolean;
begin
    θ = ∅;
    (* returns the ordered list of the body literals (null, L_1, ..., L_n, last)  *)
    B := BodyToList(R);        L := L_1;
    MatchFound := true;
    S := ∅;
    while L ≠ null
        Match(L, θ, MatchFound);
        if  MatchFound
            if(L ≠ last) ) then
                L := NextLiteral(L);
            else (* θ is a total substitution for the variables of R *)
                S := S ∪ θ;
                L := PreviousLiteral(L);
                MatchFound := false (* look for another solution *)
                θ := θ |_PreviousVars(L)
        else
            L := PreviousLiteral(L);
            θ := θ |_PreviousVars(L)
    output S;
end;
```

Fig. 1. Computing the instantiations of a rule

the quantitative information on the size of any predicate extension) that satisfies this assumption.

Instantiate first stores the body literals $L_1, \ldots, L_n$ into an ordered list $B = (null, L_1, \cdots, L_n, last)$. Then, it starts the computation of the substitutions for r. To this end, it maintains a variable θ, initially set to $\emptyset$, representing, at each step, a partial substitution for $var(r)$.

Now, the computation proceeds as follows: For each literal L_i, we denote by $PreviousVars(L_i)$ the set of variables occurring in any literal that precedes L_i in the list B, and by $FreeVars(L_i)$ the set of variables that occurs for the first time in L_i, i.e., $FreeVars(L_i) = var(L_i) - PreviousVars(L_i)$.

At each iteration of the **while** loop, we try to find a match for a literal L_i with respect to θ. More precisely, if $FreeVars(L_i) \neq \emptyset$, we look for an extension of θ to the variables in $FreeVars(L_i)$; otherwise, we simply check whether θ is a valid substitution for L_i. This is accomplished by the procedure *Match* (figure 2) that, in turns, calls *FirstMatch* if this is the first attempt to find a match for L_i, or *NextMatch* if we already have a valid substitution for L_i and we have to look for a further one.

If there is no such a substitution, then we backtrack to the previous literal in the list, or else we consider two cases: if there are further literals to be evaluated, then we continue with the next literal in the list; otherwise, θ encodes a (total) valid substitution and is thus added to the output set S. Even in this case, we backtrack for finding another solution, since we want to compute *all* instantiations of r.

Note that this kind of classical backtracking procedure works well for rules with a few literals and with a few tuples for each predicate extension. However, **DLV** has been designed to work even for manipulating complex knowledge on large databases, and for such applications the simple algorithm described above is not satisfactory.

Procedure *Match* (L:Literal, var θ:Substitution, var $MatchFound$: Boolean)
begin
  if $MatchFound$ **then**
    FirstMatch(L, θ, $MatchFound$); (* this is the first try on a new literal *)
  else (* the last match failed, look for another match on a previous literal *)
    NextMatch(L, θ, $MatchFound$);
end;

Procedure *FirstMatch* (L: Literal, var θ: Substitution, var $MatchFound$: Boolean)
(* Look in the extension I_L for the first tuple of values matching θ, and possibly update θ accordingly. The boolean variable MatchFound is assigned True if such a matching tuple has been found; otherwise, it is assigned False. *)

Procedure *NextMatch* (L: Literal, var θ: Substitution, var $MatchFound$: Boolean)
(* Similar to FirstMatch, but finds the next matching tuple. *)

Fig. 2. The matching procedures

Example 2. Suppose we want to compute all ground instantiations of the rule

$$r2: \ a(X,Y) :- p_1(X,Y), p_2(X,Z), p_3(Z,H,T), p_4(T,W), p_5(X,V,Z), p_6(X,Y,V).$$

and that we have already computed a partial substitution θ for the variables $\{X, Y, Z, H, T, W\}$, but we are not able to find a consistent value for V in the extension of p_5, in order to extend θ. In this case, according to the algorithm in Figure 1, we should backtrack to the previous literal p_4. However, the failure on atom $p_5(X,V,Z)$ is independent of variables $\{H,T,W\}$, and thus we should just find another possible value for Z. It follows that, intuitively, we could safely backtrack directly to atom $p_2(X,Z)$, where this variable has been instantiated. Thus, jumping over both $p_3(Z,H,T)$ and $p_4(T,W)$ can allow us to save a very large amount of time, especially if the extensions of p_3 and p_4 contains many tuples.

To overcome these kind troubles, a number of extensions of the backtracking technique have been described in the literature – see, e.g., the *intelligent backtracking* technique developed in the logic programming community [3], or the various backjumping techniques proposed for solving constraint satisfaction problems (CSPs) [25]. Our rule instantiation problem is closer to CSP, however most of these algorithms focused on problems with just binary constraints, and looking for just one solution. On the contrary, in our context, we need a specialized algorithm that should be able to compute efficiently all instantiations of a rule with predicates of arbitrary arity, which corresponds to finding all solutions of general (non-binary) constraint satisfaction problems.

4 A Backjumping Technique for DLP Programs Instantiation

In this section, we describe the Algorithm *BJ_Instantiate*, that given a rule r and a set of relevant variables *OutputVars*, returns a set of substitutions for these variables that are in a one-to-one correspondence with all and only the ground instances of r we are interested in. That is, we do not generate all those ground instances of r that differ only on non-relevant variables. Formally, *BJ_Instantiate* returns the projections on *OutputVars* of all the valid substitutions for r. We call these substitutions the *relevant solutions* of our problem.

The basic schema of this algorithm is no more the classical backtracking paradigm, but rather a structure-based backjumping paradigm, well studied in the constraint satisfaction area (see., e.g., [8, 25]). In these kind of algorithms, when some backtrack

```
Algorithm BJ_Instantiate
Input  R: Rule, I: Set of instances for the predicates occurring in B(R),
       OutputVars: Set of Variables;
Output S: Set of Substitutions;
var  L: Literal, B: List of Atoms, θ: Substitution, CSB: Literal, Status: MATCH_STATUS;
begin
    θ = ∅;
    (* returns the ordered list of the body literals (null, L_1, ..., L_n, last)  *)
    B := BodyToList(R);
    L := L_1;
    Status := SuccessfulMatch;
    CSB := null;
    S := ∅;
    while L ≠ null
        Match(L, θ, Status);
        switch (Status)
            case SuccessfulMatch
                if(L ≠ last) ) then
                    L := NextLiteral(L);
                else (* θ is a total substitution for the variables of R *)
                    S := S ∪ θ |_OutputVars;
                    L := BackFromSolutionFound(L, CSB, Status);
                    θ := θ |_PreviousVars(L)
                break;
            case FailureOnFirstMatch
                L := BackFromFailureOnFirstMatch(L, CSB);
                θ := θ |_PreviousVars(L)
                break;
            case FailureOnNextMatch
                L := BackFromFailureOnNextMatch(L, CSB);
                θ := θ |_PreviousVars(L)
                break;
    output S;
end;
```

Fig. 3. The algorithm BJ_Instantiate

step is necessary, it is possible to jump more than one element, rather than just one, as in the standard chronological algorithm. Of course, such jumps should be designed carefully, in order to avoid that some solution is missed, especially in our case, where we have to compute all solutions.

Let r be a rule and B the ordered list of its body literals $(null, L_1, \cdots, L_n, last)$. We say that L_i $(1 \leq i \leq n)$ is a *binder* for a variable X if there is no literal L_j, with $1 \leq j < i$ such that $X \in var(L_j)$. Moreover, for a set of variables V and a literal L_k, let *ClosestBinder*(L_k,V) denote the greatest literal L_i among the binders of the variables in V. A crucial notion in our algorithm is the *Closest Successful Binder* (CSB), which represents, intuitively, the greatest literal that is a binder of some variable X whose current assigned value belongs to the last computed solution. The CSB acts as a barrier for some kind of jumps, as described later in this section.

Another important point is the structure of the relationships among the literals in the body. We say that, for any pair of literals L_i, L_j in B, $L_i \prec_d L_j$ if $i \leq j$ and $var(L_i) \cap var(L_j) \neq \emptyset$. Let $\prec$ denote the transitive closure of the $\prec_d$ relationship and, for any literal L in B, let $dep(L) = \bigcup_{\{L' | L \prec L'\}} var(L')$. Intuitively, this is the set of variables that depends on the instantiation of the literal L, and we refer to it as the *dependency set* of L.

Example 3. As a running example in this section, consider the following rule

$$r_3 : a(X, Y, Z) :- q_1(X, T, W), q_2(X, Y), q_3(Z, S), q_4(Z, V), q_5(T, H), q_6(H, T, V).$$

enum MATCH_STATUS = {SuccessfulMatch, FailureOnFirstMatch, FailureOnNextMatch};

Procedure *Match* (*L*:Literal, var θ:Substitution, var *Status*: MATCH_STATUS)
begin
  if *Status* = SuccessfulMatch **then** (* the last match was successful *)
    FirstMatch(*L*, θ, *Status*); (* this is the first try on a new literal *)
  else (* the last match failed, look for another match on a previous literal *)
    NextMatch(*L*, θ, *Status*);
end;

Procedure *FirstMatch* (*L*: Literal, var θ: Substitution, var *Status*: MATCH_STATUS)
(* Look in the extension I_L for the first tuple of values matching θ, and possibly update θ accordingly. *Status* is assigned SuccessfulMatch if such a matching tuple exists; otherwise, it is assigned FailureOnFirstMatch *)

Procedure *NextMatch* (*L*: Literal, var θ: Substitution, var *Status*: MATCH_STATUS)
(* Similar to FirstMatch, but finds the next matching tuple. In case of failure, *Status* is set to FailureOnNextMatch *)

Fig. 4. Matching procedures for *BJ_Instantiate*

It is easy to check that the dependency set of literal $q_5(T, H)$ is $\{T, H, V\}$, while the dependency set of $q_3(Z, S)$ is $\{Z, S, V, H, T\}$.

In order to instantiate r_3, our algorithm needs the additional information on the relevant variables and the already known instances for the predicates occurring in the body. Then, assume that $OutputVars = \{X, Y, Z, T, W\}$, and that we are given the following extensions for the predicates occurring in $B(r_3)$:

$q_1(x_1, t_1, w_1)$	$q_2(x_1, y_1)$	$q_3(z_1, s_1)$	$q_4(z_1, v_1)$	$q_5(t_2, h_1)$	$q_6(h_2, t_2, v_1)$
$q_1(x_1, t_2, w_1)$	$q_2(x_1, y_2)$		$q_4(z_1, v_2)$	$q_5(t_2, h_2)$	$q_6(h_2, t_2, v_2)$

Figure 3 shows the algorithm *BJ_Instantiate*. As for Algorithm *Instantiate*, at each iteration of the **while** loop, the procedure *Match* tries to find a match for a literal L_i with respect to the current partial substitution θ. If it succeeds and L_i is not the last literal, then we can proceed with the next literal L_{i+1}. Otherwise, we have to backtrack, and thus we have to decide where to jump and, possibly, update the current CSB. Now, we have a number of different cases to be handled, depending on the outcome *Status* of the procedure *Match*.

1. **Success, and θ encodes a total substitution**. Since also the match on the last literal is successful, θ encodes a valid substitution for the variables in r, and its restriction to *OutputVars* is therefore added to the set of solutions. Then, in order to look for further solutions, we have to backtrack. However, in this algorithm, we are not forced to go back to the previous literal. Rather, we can jump to the closest literal L_j binding a variable of interest, that is, jump to *ClosestBinder*(*last*,*OutputVars*). Moreover, in this case the CSB is set to L_j.

   Example 4. In our running example, the algorithm is able to find the total substitution $\theta(X) = x_1$, $\theta(Y) = y_1$, $\theta(Z) = z_1$, $\theta(T) = t_2$, $\theta(W) = w_1$, $\theta(S) = s_1$, $\theta(V) = v_1$, and $\theta(H) = h_2$. That is, we have a match for all the literals in B and we are at *last*. Then, the restriction of θ to the set of relevant variables is added to S. In our case, this solution corresponds to the following instance of r_3:

   $$a(x_1, y_1, z_1) :- q_1(x_1, t_2, w_1), q_2(x_1, y_1), q_3(z_1, s_1).$$

   Now, according to the algorithm, we jump back to $q_3(Z, V)$ for finding other solutions. Note that we do not look for further consistent tuples in the extensions

```
Function BackFromFailureOnFirstMatch (L: literal, var CSB: Literal) : Literal;
begin (* the first match on a new literal failed *)
    L' := ClosestBinder(L, Vars(L));
    if L' ≺ CSB then
        CSB := L';
    return L';
end;

Function BackFromFailureOnNextMatch(L: Literal, var CSB: Literal) : Literal;
begin (* failure looking for another match for L *)
    if L = CSB then
        CSB := ClosestBinder(L,OutputVars);
    L' := ClosestBinder(L,DepVars(L))
    return max_≺{L', CSB};
end;

Function BackFromSolutionFound(L: Literal, var CSB: Literal, var Status: MATCH_STATUS)
        : Literal;
begin
    Status := FailureOnNextMatch; (* look for another solution *)
    CSB := ClosestBinder(L,OutputVars);
    return CSB;
end;
```

Fig. 5. Backjumping procedures for *BJ_Instantiate*

of q_4, q_5, and q_6, because they do not bind any relevant variable. Indeed, possible solutions coming from other instances of these predicates (e.g., the solution with $\theta(V) = v_2$) would just lead to useless rules in the instantiation of the program at hand. Finally, the CSB is set to q_3.

2. **Failure at the first attempt to find a match for a literal** L_i. We jump back to the closest literal L_j binding any of the variables in L_i, that is, jump to *ClosestBinder*(L_i,$var(L_i)$). Indeed, in this case, the only way for finding a match for L_i is to change the assignment for some of its bound variables. Moreover, if L_j precedes CSB, then we can push back CSB to L_j. This will make the next type of jumps less restrictive, see case 3 below.

   Example 5. In our running example, the first time that we try to find a match for q_5, we have computed the partial substitution $\theta(X) = x_1$, $\theta(Y) = y_1$, $\theta(Z) = z_1$, $\theta(T) = t_1$, $\theta(W) = w_1$, and $\theta(S) = s_1$. In this case, we are not able to find any matching instance in the extension of q_5. Indeed, none of its instances has a value t_1 for variable T. Then, we have to change the value assigned to one of the variables occurring in q_5, and thus we can safely jump over q_4, q_3, and q_2, and try to match again $q_1(X, T, W)$. Indeed, q_1 is the closest binder for $var(q_5)$, as it determines the value for variable T.

3. **Failure while looking for another match for a literal** L_i. In this case, L_i is a binder of some set of variables $\bar{X}$, and we fail in finding a different consistent substitution for these variables. Since we were successful on our first attempt to deal with L_i, this means that, for some reason, we jumped back to L_i from some later item, say L_j, of the list B. Now, we have to decide where to jump after the current failure, and this time the variables occurring in L_i are not the only candidates to be changed. Rather, we have to look at the dependency set of L_i, as shown below.

   Example 6. Assume that, in our running example, we are looking for another match for $q_3(Z, S)$ and that the CSB is set to $q_1(X, T, W)$. According to the

algorithm, we have to jump to q_1, even if it is not a binder for any variable occurring in q_3. The reason is that q_1 is a binder for T, which belongs to the dependency set of q_3, and changing its value may lead to some new solution (possibly comprising values already considered for the variables occurring in q_3).

Another important issue concerns the management of the CSB. First, we check whether the current literal L_i coincides with the CSB. If this is the case, we push back the CSB to *ClosestBinder*(L_i,$OutputVars$). In this case, it acts as a barrier and cannot be jumped, otherwise we can miss some relevant solution as the following example shows.

Example 7. Let us continue from the execution step described at point 1 above, where we have found our first solution. Recall that we jumped back to $q_3(Z, S)$ and the CSB is set to this literal. In this case, the CSB is first pushed back to $q_2(X, Y)$, which is the *ClosestBinder*($q_3(Z, S)$,$OutputVars$). Then, even if according to the dependency set we could jump to q_1, we are forced to stop our jumping back to literal $q_2(X, Y)$, because of the CSB limit. It is worthwhile noting that, if we go directly to $q_1(X, T, W)$, we miss the solution obtainable by assigning y_2 to variable Y and corresponding to the following instance of r_3:

$$a(x_1, y_2, z_1) :- q_1(x_1, t_2, w_1), q_2(x_1, y_2), q_3(z_1, s_1).$$

Theorem 1. *Algorithm* BJ_Instantiate *is sound and complete. That is, given a rule r, the ground instances for the predicates occurring in its body, and the set of its relevant variables OutputVars,* BJ_Instantiate *computes the set containing all and only the projections over OutputVars of the valid substitutions for r.*

5 Experimental Results

5.1 Benchmark Programs

In order to check the validity of the proposed method, we have implemented it in the grounding engine of the **DLV** system, and we have run it on a collection of benchmark programs taken from different domains. For space limitation, we do not include the code of benchmark programs. However, we give below a very short description of the problems that are encoded in these benchmark programs.

RAMSEY(3,6) $\neq$ 17 Prove that 17 is not the Ramsey number $Ramsey(3, 6)$ [21].

CONSTRAINT-3COL[20,30] A one-rule encoding of 3-colorability (the classical encoding of 3-colorability as a constraint satisfaction problem), on a graph with 20 nodes and 30 edges.

CONSTRAINT-3COL[30,40] Similar to the previous one, but on a graph with 30 nodes and 40 edges.

CONSTRAINT-5COL[20,30] Again, a one-rule encoding of colorability, but for 5 colors and on a graph with 20 nodes and 30 edges.

CRISTAL Deductive databases application that involves complex knowledge manipulations on databases, developed at CERN in Switzerland.

SCHEDULING A scheduling program for determining shift rotation of employees.

HANOI[6discs,63steps] Hanoi Towers with 6 discs and 63 steps.

ANCESTOR Given a parent relationship, find the genealogy tree of each person in the database.

K-DECOMP Decide whether there exists a hypertree decompositon [16] of a conjunctive query having width$\leq$K.

5.2 Backtracking vs the new BackJumping Technique

We provided a C++ implementation of Algorithm *BJ_Instantiate* and we integrated it with the rest of the Instantiator module in the **DLV** system. Then, we run a number of experiments by using the above benchmark problems, in order to compare the performances of the previous backtracking-based rule instantiator with the new one, proposed in this paper. All binaries were produced by the GNU compiler GCC 2.95.2, and the experiments were performed on a Sun UltraSparc2 machine.

Table 1 shows the results of our tests. For each benchmark program P described in column 1, column 2 (respectively, 3) reports the times employed to instantiate P by using **DLV**, when Algorithm *Instantiate* (resp., Algorithm *BJ_Instantiate*) is used in the rule instantiator module. All running times are expressed in seconds.

Program	Backtracking	Backjumping
RAMSEY(3,6)$\neq$ 17	172.10s	44.91s
CONSTRAINT-3COL[20,30]	10.08s	0.00s
CONSTRAINT-3COL[40,50]	188.24s	0.00s
CONSTRAINT-5COL[20,30]	–	0.00s
CRISTAL	43.30s	39.20s
SCHEDULING	76.20s	75.25s
HANOI[6discs,63steps]	12.79s	12.88s
ANCESTOR	26.31s	25.90s
K-DECOMP	60.70s	59.64s

Table 1. A comparison between the backtracking and the backjumping techniques

The results confirm the intuition that the new backjumping-based procedure outperforms the previous one in many cases, and can be very useful for improving the efficiency of **DLV** (and of any other ASP system that could exploit its instatiator).

Of course, the speed-up is not that high if we have to instantiate programs where all rules are very short, and where thus the two procedures exhibit a similar behavior. Note that, in some cases, the old procedure can be also slightly better than the new one, since the latter has some overhead due to the computation of the dependency sets and the management of the CSB. This is witnessed, e.g., by HANOI[6discs,63steps].

However, we have an impressive speed-up when programs contain some rules with many literals in their bodies and/or when such rules have a few relevant variables (i.e., many solved predicates occur in their bodies). For instance, CONSTRAINT-5COL[20,30] consists of a single long rule where all predicates are solved. In fact, in this extreme case, we stopped the old procedure that was still running after 2000s, while the new one instantiated the program almost instantaneously. Moreover, note that we may have a very good speed-up even if all variables are relevant, as witnessed by the program RAMSEY(3,6)$\neq$ 17.

Currently, our experimentation activity continues on further benchmark problems. Also, we are evaluating the quality of the ground programs computed by the algorithm, as well as the impact of such instantiations on the overall system performance.

Acknowledgments

The authors are grateful to Stefania Galizia and Nicola Leone for several useful discussions on the algorithm and its implementation.

This work was supported by the European Commission under project INFOMIX, project no. IST-2001-33570, and under project ICONS, project no. IST-2001-32429.

References

1. C. Anger, K. Konczak, and T. Linke. `NoMoRe`: A System for Non-Monotonic Reasoning. In Proc. of LPNMR'01, pp. 406–410, LNAI 2173, Springer, 2001.
2. C. Aravindan, J. Dix, and I. Niemelä. DisLoP: A Research Project on Disjunctive Logic Programming. *AI Communications*, 10(3/4):151–165, 1997.
3. M. Bruynooghe, and L. Pereira. Deduction revision by intelligent backtracking *Implementations of Prolog* (J. Campbell, ed.), Ellis Horwood, 1984, pp. 194-215.
4. Y. Babovich. Cmodels homepage, since 2002. `http://www.cs.utexas.edu/users/tag/cmodels.html`.
5. Weidong Chen and David Scott Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE TKDE*, 8(5):742–757, 1996.
6. Paweł Cholewiński, V. Wiktor Marek, Artur Mikitiuk, and Mirosław Truszczyński. Computing with Default Logic. *Artificial Intelligence*, 112(2–3):105–147, 1999.
7. Paweł Cholewiński, V. Wiktor Marek, and M. Truszczyński. Default Reasoning System DeReS. In *Proc. of KR '96*, pp. 518–528, 1996.
8. R. Dechter. Enhancement schemes fo constraint processing:backjumping, learning and cutset decomposition. *Artificial Intelligence*, 41, 1990.
9. D. East and M. Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proc. of KI'2001*, pp. 138–153, LNAI 2174, Springer, 2001.
10. D. East and M. Truszczyński. dcs: An Implementation of DATALOG with Constraints. *Proc. of NMR'2000*, April 2000.
11. D. East and M. Truszczyński. System Description: aspps – An Implementation of Answer-Set Programming with Propositional Schemata. In *proc. of LPNMR'01*,pp. 402–405, LNAI 2173, Springer, 2001.
12. U. Egly, T. Eiter, H. Tompits, and S. Woltran.Solving Advanced Reasoning Tasks using Quantified Boolean Formulas. In *Proc. of AAAI'00*, pp. 417–422. AAAI Press / MIT Press, 2000.
13. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and Francesco Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. *Proc. of KR'98*, pp. 406–417, 1998.
14. W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *Proceedings of the 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99)*, pp. 135–139, 1999.
15. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
16. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *JCSS*, 64(3):579-627, 2002.
17. V. Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm – A 25-Year Perspective*, pp. 357–373. Springer, 1999.
18. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proc. of AAAI-2002*, AAAI Press / MIT Press, 2002.
19. N. McCain and H. Turner. Satisfiability Planning with Causal Theories. *Proc. of KR'98*, pp. 212–223, 1998.
20. Teodor C. Przymusinski. Stable Semantics for Disjunctive Programs. *New Generation Computing*, 9:401–424, 1991.
21. S. Radziszowski. Small Ramsey Numbers. *The Electronic J. of Combinatorics*, 1, 1999.
22. P. Rao, K.F. Sagonas, T. Swift, D.S. Warren, and J. Freire. XSB: A System for Efficiently Computing Well-Founded Semantics. *Proc. of LPNMR'97*, pp.2–17, LNAI 1265, Springer, 1997.
23. D. Seipel and H. Thöne. DisLog – A System for Reasoning in Disjunctive Deductive Databases. *Proc. of DAISD'94*, pp. 325–343, 1994.
24. P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, 2002.
25. Tsang, E.P.K. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
26. J. D. Ullman. *Principles of Database and Knowledge Base Systems*, Computer Science Press, 1989.