Look-Back Techniques and Heuristics in DLV: Implementation and Evaluation*

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

[1] Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber,leone,maratea,ricca}@mat.unical.it
[2] DIST, University of Genova, 16145 Genova, Italy
marco@dist.unige.it

Abstract. Answer Set Programming (ASP) is a purely-declarative programming paradigm based on logic rules, allowing for both disjunction in the head of the rules and nonmonotonic negation in the body. ASP can express any property whose complexity is in the second level of the polynomial hierarchy, thus it is strictly more powerful than propositional logic under standard complexity conjectures. DLV is the state-of-the-art *disjunctive* ASP system, and it is based on an algorithm using backtracking search, like the vast majority of the currently available ASP systems. Despite its efficiency, until recently, DLV did not incorporate any "backjumping" techniques (neither did other disjunctive ASP systems). Related, DLV could not use "look-back" information accumulated for backjumping in its heuristics, which have been shown in related research areas to be crucial on large benchmarks stemming from applications. In this paper, we focus on the experimental evaluation of the look-back algorithms and heuristics that have been implemented in DLV. We have conducted a wide experimental analysis considering both randomly-generated and structured instances of the 2QBF problem (the canonical problem for the complexity classes Σ_2^P and Π_2^P). The results show that the new look-back techniques significantly improve the performance of DLV, being performance-wise competitive even with respect to "native" QBF solvers.

1 Introduction

Answer Set Programming (ASP) [1, 2] is a purely-declarative programming paradigm based on nonmonotonic reasoning and logic programming. The idea of answer set programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver to find such solutions [3]. The language of ASP is very expressive, allowing for both disjunction in the head of the rules and nonmonotonic negation in the body, and able to represent every property in the second level of the polynomial hierarchy. Therefore, ASP is strictly more powerful than propositional logic unless $P = NP$.

DLV is the state-of-the-art *disjunctive* ASP system, and it is based on an algorithm relying on backtracking search, like most other competitive ASP systems. Until recently,

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

DLV did not incorporate any "look-back" techniques, like "backjumping" procedures and "look-back" heuristics. By "backjumping" [4] we refer to an optimized recovery upon inconsistency during the search where, instead of restoring the state of the search up to the previous choice and then "flipping" its value, we try to "jump over" choices that are not relevant for the inconsistency we met. This is done by means of a reason calculus, which records information about the literals ("reasons") whose truth has caused the truth of other derived literals.

Look-back heuristics [5] further strengthen the potential of backjumping by using the information made explicit by the reasons. The idea of such family of heuristics is to preferably choose atoms which frequently caused inconsistencies, thus focusing on "critical" atoms. This significantly differ from classical ASP heuristics that use information arising from the simplification part ("look-head") of the algorithm. Such look-back optimization techniques and heuristics have been shown, on other research areas, to be very effective on "big" benchmarks coming from applications, like planning and formal verification.

In this paper, we report on the analysis, implementation and evaluation of the backjumping technique and look-back heuristics in DLV, and ultimately, about their efficiency in the disjunctive ASP setting. Such methods have been already used in other ASP systems which (i) do not allow for disjunction in the head of the rules [6, 7], or (ii) apply such methods only indirectly after a transformation to a propositional satisfiability problem [8]. The resulting system, called DLV^{LB}, is therefore the first implementation of disjunctive ASP featuring backjumping and look-back heuristics. Importantly, our system provides several options regarding the initialization of the heuristics and the truth value to be assigned to an atom chosen by the heuristics. In our experimental analysis, we provide a comprehensive comparison of the impact of these options, and demonstrate how the new components of DLV^{LB} enhances the efficiency of DLV. Moreover, we also provide a comparison to the other competitive disjunctive ASP systems GnT and Cmodels, which are generally outperformed considerably by DLV^{LB} on the considered benchmarks. Finally, we also present a comparison with respect to QBF solvers, which also allow for solving problems within the second level of the polynomial hierarchy.

2 Answer Set Programming Language

A *(disjunctive) rule* r is a formula

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \texttt{not } b_{k+1}, \cdots, \texttt{not } b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are function-free atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while $b_1, \cdots, b_k, \texttt{not } b_{k+1}, \cdots, \texttt{not } b_m$ is the *body*, of which $b_1, \cdots, b_k$ is the *positive body*, and $\texttt{not } b_{k+1}, \cdots, \texttt{not } b_m$ is the *negative body* of r.

An *(ASP) program* $\mathcal{P}$ is a finite set of rules. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables. Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_{\mathcal{P}}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $Ground(\mathcal{P})$ of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, its answer sets are defined using its ground instantiation $Ground(\mathcal{P})$ in two steps: First answer sets of positive programs are defined, then a reduction of general programs to positive ones is given, which is used to define answer sets of general programs. A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\texttt{not}\ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.[3] A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. Interpretation I is *total* if, for each atom A in $B_{\mathcal{P}}$, either A or $\texttt{not}\ A$ is in I (i.e., no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is an *answer set* for a positive program $\mathcal{P}$ if it is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all rules $r \in \mathcal{P}$ the negative body of which is false w.r.t. X and (ii) deleting the negative body from the remaining rules. An answer set of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

3 Answer Set Computation Algorithms

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, which will be used for the experiments, but also other ASP systems, employ a similar procedure. In general, an answer set program $\mathcal{P}$ contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[4] The subsequent computations, which constitute the non-deterministic hearth of the system, are then performed on $ground(\mathcal{P})$ by the so called Model Generator procedure.

In the following paragraphs, we illustrate the original model generation algorithm of DLV (which is based on chronological backtracking); then, we briefly describe a backjumping technique that has been implemented in the system[10]; and, we detail how the model generation algorithm has been changed to introduce it. Finally, we report a description of all the heuristics, including the new ones based look-back techniques, that have been implemented in the DLV system, so far.

The Standard Model Generator Algorithm. The computation of answer sets is performed by exploiting the Model Generator Algorithm sketched in Figure 1.[5]

[3] We represent interpretations as set of literals, since we have to deal with partial interpretations in the next sections.

[4] Note that $ground(\mathcal{P})$ is usually not the full $Ground(\mathcal{P})$; rather, it is a subset (often much smaller) of it having precisely the same answer sets as $\mathcal{P}$ [9]

[5] Note that for reasons of presentation, the description here is quite simplified w.r.t. the "real" implementation. A more detailed description can be found in [11].

```
bool ModelGenerator ( Interpretation& I ) {
    I = DetCons ( I );
    if ( I == ℒ ) then
      return false;
    if ( "no atom is undefined in I" ) then return IsAnswerSet(I);
    Select an undefined atom A using a heuristic;
    if ( ModelGenerator ( I ∪ {A} ) then return true;
      else return ModelGenerator ( I ∪ {not A} ); };
```

Fig. 1. Computation of Answer Sets

This function is initially called with parameter I set to the empty interpretation.[6]

If the program $\mathcal{P}$ has an answer set, then the function returns True, setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the DPLL procedure employed by SAT solvers. It first calls a function DetCons, which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom A is crucial for the performance of an ASP system. In the following, we will describe some heuristic criteria for the selection of such branching atoms.

If no atom is left for branching, the Model Generator has produced a "candidate" answer set, the stability of which is subsequently verified by *IsAnswerSet(I)*. This function checks whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I, and outputs the model, if so. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise.

Note that the algorithm described above computes one answer set for simplicity, however it can be straightforwardly modified to compute all or n answer sets.

Backjumping and Reason for Literals. If during the execution of the ModelGenerator function described in previous paragraph a contradiction arises, or the stable model candidate is not a minimal model, ModelGenerator backtracks and modifies the last choice. This kind of backtracking is called chronological backtracking.

We now describe a technique in which the truth value assignments causing a conflict are identified and backtracking is performed "jumping" directly to a point so that at least one of those assignments is modified. This kind of backtracking technique is called non-chronological backtracking or backjumping. To give the intuition on how backjumping is supposed to works, we exploit the following example.

Consider the program of Figure 2(a) and suppose that the search tree is as depicted in Figure 2(b).

According to this tree, we first assume a to be true, deriving b to be false (from r_1 to ensure the minimality of answer sets). Then we assume c to be true, deriving d to be false

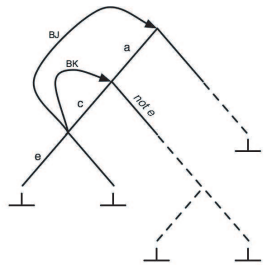
[6] Observe that the interpretations built during the computation are 3-valued, that is, a literal can be True, False or Undefined w.r.t. I.

$r_1: \ a \vee b. \quad r_2: \ c \vee d. \quad r_3: \ e \vee f.$

$r_4: \ g :- a, e. \quad r_5: \ :- g, a, e.$

$r_6: \ g :- a, f. \quad r_7: \ :- g, a, f.$

(a)



(b)

Fig. 2. Backtracking vs Backjumping.

(from r_2 for minimality). Third, we assume e to be true and derive f to be false (from r_3 for minimality) and g to be true (from r_4 by forward inference). This truth assignment violates constraint r_5 (where g must be false), yielding an inconsistency. We continue the search by inverting the last choice, that is, we assume e to be false and we derive f to be true (again from r_3 to preserve minimality) and g to be true (from r_6 by forward inference), but obtain another inconsistency (because constraint r_7 is violated, here g must also be false).

At this point, ModelGenerator goes back to the previous choice point, in this case inverting the truth value of c (cf. the arc labelled BK in Fig. 2(b)).

Now it is important to note that the inconsistencies obtained are independent of the choice of c, and only the truth value of a and e are the "reasons" for the encountered inconsistencies. In fact, no matter what the truth value of c is, if a is true then any truth assignment for e will lead to an inconsistency. Looking at Fig. 2(b), this means that in the whole subtree below the arc labelled a no stable model can be found. It is therefore obvious that the chronological backtracking search explores branches of the search tree that cannot contain a stable model, performing a lot of useless work.

A better policy would be to go back directly to the point at which we assumed a to be true (see the arc labelled BJ in Fig. 2(b)). In other words, if we know the "reasons" of an inconsistency, we can backjump directly to the closest choice that caused the inconsistent subtree. In practice, once a literal has been assigned a truth value during the computation, we can associate a reason for that fact with the literal. For instance, given a rule $a :- b, c, \texttt{not}\ d.$, if b and c are true and d is false in the current partial interpretation, then a will be derived as true (by Forward Propagation). In this case, we can say that a is true "because" b and c are true and d is false. A special case are *chosen* literals, as their only reason is the fact that they have been chosen. The chosen literals can therefore be seen as being their own reason, and we may refer to them as elementary reasons. All other reasons are consequences of elementary reasons, and hence aggregations of elementary reasons. Each literal l derived during the propagation (i.e., DetCons) will have an associated set of positive integers $R(l)$ representing the reason of l, which are essentially the recursion levels of the chosen literals which entail l. Therefore, for any chosen literal c, $|R(c)| = 1$ holds. For instance, if $R(l) = \{1, 3, 4\}$, then the literals chosen at recursion levels 1, 3 and 4 entail l. If $R(l) = \emptyset$, then l is true in all answer sets.

The process of defining reasons for derived (non-chosen) literals is called *reason calculus*. The reason calculus we employ defines the auxiliary concepts of satisfying literals

and orderings among satisfying literals for a given rule. It also has special definitions for literals derived by the well-founded operator. Here, for lack of space, we do not report details of this calculus, and refer to [10] for a detailed definition.

When an inconsistency is determined, we use reason information in order to understand which chosen literals have to be undone in order to avoid the found inconsistency. Implicitly this also means that all choices which are not in the reason do not have any influence on the inconsistency. We can isolate two main types of inconsistencies: (i) Deriving conflicting literals, and (ii) failing stability checks. Of these two, the second one is a peculiarity of disjunctive ASP.

Deriving conflicting literals means, in our setting, that DetCons determines that an atom a and its negation $\texttt{not}\ a$ should both hold. In this case, the reason of the inconsistency is – rather straightforward – the combination of the reasons for a and $\texttt{not}\ a$: $R(a) \cup R(\texttt{not}\ .a)$.

Inconsistencies from failing stability checks are a peculiarity of disjunctive ASP, as non-disjunctive ASP systems usually do not employ a stability check. This situation occurs if the function IsAnswerSet(I) of ModelGenerator returns false, hence if the checked interpretation (which is guaranteed to be a model) is not stable. The reason for such an inconsistency is always based on an unfounded set, which has been determined inside IsAnswerSet(I) as a side-effect. Using this unfounded set, the reason for the inconsistency is composed of the reasons of literals which satisfy rules which contain unfounded atoms in their head (the cancelling assignments of these rules). The information on reasons for inconsistencies can be exploited for backjumping by going back to the closest choice which is a reason for the inconsistency, rather than always to the immediately preceding choice.

In the next paragraph, we will describe a modified version of the ModelGenerator algorithm which implements the above-sketched backjumping technique.

The Model Generation Algorithm with Backjumping. In this paragraph we describe ModelGeneratoBJ (shown in Fig. 3) a modification of the ModelGenerator function, which is able to perform non-chronological backtracking.

It extends ModelGenerator by introducing additional parameters and data structures, in order to keep track of reasons and to control backtracking and backjumping. In particular, two new parameters IR and bj_level are introduced, which hold the reason of the inconsistency encountered in the subtrees of which the current recursion is the root, and the recursion level to backtrack or backjump to. When going forward in recursion, bj_level is also used to hold the current level.

The variables $curr_level$, $posIR$, and $negIR$ are local to ModelGeneratoBJ and used for holding the current recursion level, and the reasons for the positive and negative recursive branch, respectively.

Initially, the ModelGeneratorBJ function is invoked with I set to the empty interpretation, IR set to the empty reason, and bj_level set to -1 (but it will become 0 immediately). Like the ModelGenerator function, if the program $\mathcal{P}$ has an answer set, then the function returns true and sets I to the computed answer set; otherwise it returns false. Again, it is straightforward to modify this procedure in order to obtain all or up to n answer sets. Since these modification gives no additional insight, but rather obfuscates the main technique, we refrain from presenting it here.

```
bool ModelGeneratorBJ (Interpretation& I, Reason& IR,
                       int& bj_level ) {
    bj_level ++;
    int curr_level = bj_level;
    I = DetConsBJ ( I, IR );
    if ( I == ℒ ) return false;
    if ( "no atom is undefined in I" )
      if IsAnswerSetBJ( I, IR ); return true;
      else
        bj_level = MAX ( IR );
        return false;
    Reason posIR, negIR;
    Select an undefined atom A using a heuristic;
    R(A)= { curr_level };
    if ( ModelGeneratorBJ( I ∪ {A}, posIR, bj_level ) return true;
    if (bj_level < curr_level)
      IR = posIR; return false;
    bj_level = curr_level;
    R(not A) = { curr_level };
    if ( ModelGeneratorBJ ( I ∪ {not A}, negIR, bj_level ) return true;
    if ( bj_level < curr_level )
      IR = negIR; return false;
    IR = trim( curr_level, Union ( posIR, negIR ) );
    bj_level = MAX ( IR );
    return false;
};
```

Fig. 3. Computation of Answer Sets with Backjumping

ModelGeneratorBJ first calls DetConsBJ, an enhanced version of the DetCons procedure. In addition to DetCons, DetConsBJ computes the reasons of the inferred literals, as pointed out in the paragraph for reasons. Moreover, if at some point an inconsistency is detected (i.e. the complement of a true literal is inferred to be true), DetConsBJ (returns the set of all literals $\mathcal{L}$, and) builds the reason of this inconsistency and stores it in its new, second parameter IR. If an inconsistency is encountered, ModelGeneratorBJ immediately returns false and no backjumping is done. This is an optimization, because it is known that the inconsistency reason will contain the previous recursion level. There is therefore no need to analyze the levels.

If no undefined atom is left, ModelGeneratorBJ invokes IsAnswerSetBJ, an enhanced version of IsAnswerSet. In addition to IsAnswerSet, IsAnswerSetBJ computes the inconsistency reason in case of a stability checking failure, and sets the second parameter IR accordingly. If this happens, it might be possible to backjump, and we set bj_level to the maximal level of the inconsistency reason (or 0 if it is the empty set) before returning from this instance of ModelGeneratorBJ. Indeed, the maximum level in IR corresponds to the nearest (chronologically) choice causing the failure. If the stability check succeeded, we just return true.

Otherwise, an atom A is selected according to a heuristic criterion. We set the reason of A to be the current recursion level and invoke ModelGeneratorBJ recursively, using $posIR$ and bj_level to be filled in case of an inconsistency. If the recursive call returned true, ModelGeneratorBJ just returns true as well. If it returned false, the corresponding branch is inconsistent, $posIR$ holds the inconsistency reason and bj_level the recursion level to backtrack or backjump to.

Now, if bj_level is less than the current level, this indicates a backjump, and we return from the procedure, setting the inconsistency reason appropriately before. If not, then we have reached the level to go to. We set the reason for `not` A, and enter the second recursive invocation, this time using $negIR$ and reusing bj_level (which is reinitialized before). As before, if the recursive call returns true, ModelGeneratorBJ immediately returns true also, while if it returned false, we check whether we backjump, setting IR and immediately returning false. If no backjump is done, this instance of ModelGeneratorBJ is the root of an inconsistent subtree, and we set its inconsistency reason IR to the union of $posIR$ and $negIR$, deleting all (irrelevant) integers which are greater or equal than the current recursion level (this is done by the function trim). We finally set bj_level to the maximum of the obtained inconsistency reason (or 0 if the set is empty) and return false.

The actual implementation in DLV is slightly more involved, but only due to technical details. Since we do not believe that these technical issues give any particular insight, but are instead rather lengthy in description, we have opted to not include them.

The information provided by reasons can be further exploited in a backjumping-based solver. In particular, in the following paragraph we describe how reasons for inconsistencies can be exploited for defining look-back heuristics.

Heuristics. In this paragraph we will first describe the two main heuristics for DLV (based on look-ahead), and subsequently define several new heuristics based on reasons (or based on look-back), which are computed as side-effects of the backjumping technique. We assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. We first recall the "standard" DLV heuristic h_{UT} [12], which has recently been refined to yield the heuristic h_{DS} [13], which is more "specialized" for hard disjunctive programs (like 2QBF). These are look-ahead heuristics, that is, the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the interpretation resulting from the application of DetCons on $I \cup \{Q\}$; w.l.o.g., we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

Standard Heuristic of DLV (h_{UT}). This heuristic, which is the default in the DLV distribution, has been proposed in [12], where it was shown to be very effective on many relevant problems. It exploits a peculiar property of ASP, namely *supportedness*: For each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, h_{DS} is geared towards choosing those literals which minimize the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule. The heuristic h_{UT} is "balanced", that is, the heuristic values of an atom Q depends on both the effect of taking Q and `not` Q, the decision between Q and `not` Q is based on the UT atoms criteria.

Enhanced Heuristic of DLV (h_{DS}). The heuristic h_{DS} [14] is based on h_{UT}, and is different from h_{UT} only for pairs of literals which are not ordered by h_{UT}. The idea of the additional criterion is that interpretations having a "higher degree of supportedness" are preferred, where the degree of supportedness is the average number of supporting rules for the true atoms. Intuitively, if all true atoms have many supporting rules in a model M, then the elimination of a true atom from the interpretation would violate many rules, and it becomes less likely finding a subset of M which is a model of $\mathcal{P}^M$ (which would

disprove that M is an answer set). Interpretations with a higher degree of supportedness are therefore more likely to be answer sets. Just like h_{UT}, h_{DS} is "balanced".

The Look-back Heuristics (h_{LB}). We next describe a family of new look-back heuristics h_{LB}. Different to h_{UT} and h_{DS}, which provide a partial order on potential choices, h_{LB} assigns a number ($V(L)$) to each literal L (thereby inducing an implicit order). This number is periodically updated using the inconsistencies that occurred after the most recent update. Whenever a literal is to be selected, the literal with the largest $V(L)$ will be chosen. If several literals have the same $V(L)$, then negative literals are preferred over positive ones, but among negative and positive literals having the same $V(L)$, the ordering will be random. In more detail, for each literal L, two values are stored: $V(L)$, the current heuristic value, and $I(L)$, the number of inconsistencies L has been a reason for since the most recent heuristic value update. After having chosen k literals, $V(L)$ is updated for each L as follows: $V(L) := V(L)/2 + I(L)$. The motivation for the division (which is assumed to be defined on integers by rounding the result) is to give more impact to more recent values. Note that $I(L) \neq 0$ can hold only for literals that have been chosen earlier during the computation.

A crucial point left unspecified by the definition so far are the initial values of $V(L)$. Given that initially no information about inconsistencies is available, it is not obvious how to define this initialization. On the other hand, initializing these values seems to be crucial, as making poor choices in the beginning of the computation can be fatal for efficiency. Here, we present two alternative initializations: The first, denoted by h_{LB}^{MF}, is done by initializing $V(L)$ by the number of occurrences of L in the program rules. The other, denoted by h_{LB}^{LF}, involves ordering the atoms with respect to h_{DS}, and initializing $V(L)$ by the rank in this ordering. The motivation for h_{LB}^{MF} is that it is fast to compute and stays with the "no look-ahead" paradigm of h_{LB}. The motivation for h_{LB}^{LF} is to try to use a lot of information initially, as the first choices are often critical for the size of the subsequent computation tree. We introduce yet another option for h_{LB}, motivated by the fact that answer sets for disjunctive programs must be minimal with respect to atoms interpreted as true, and the fact that the checks for minimality are costly: If we preferably choose false literals, then the computed answer set candidates may have a better chance to be already minimal. Thus even if the literal, which is optimal according to the heuristic, is positive, we will choose the corresponding negative literal first. If we employ this option in the heuristic, we denote it by adding AF to the superscript, arriving at $h_{LB}^{MF,AF}$ and $h_{LB}^{LF,AF}$ respectively.

4 Experiments

We have implemented the above-mentioned look-back techniques and heuristics in DLV; in this section, we report on their experimental evaluation.

Compared Methods. For our experiments, we have compared several versions of DLV [15], which differ on the employed heuristics and the use of backjumping. For having a broader picture, we have also compared our implementations to the competing systems GnT and CModels3, and with the QBF solver Ssolve. The considered systems are:

- **dlv.ut**: the standard DLV system employing h_{UT} (based on look-ahead).
- **dlv.ds**: DLV with h_{DS}, the look-ahead based heuristic specialized for Σ_2^P/Π_2^P hard

disjunctive programs.
• **dlv.ds.bj**: DLV with h_{DS} and backjumping.
• **dlv.mf**: DLV with h_{LB}^{MF}.[7]
• **dlv.mf.af**: DLV with $h_{LB}^{MF,AF}$.
• **dlv.lf**: DLV with h_{LB}^{LF}.
• **dlv.lf.af**: DLV with $h_{LB}^{LF,AF}$.
• **gnt** [16]: The solver GnT, based on the Smodels system, can deal with disjunctive ASP. One instance of Smodels generates candidate models, while another instance tests if a candidate model is stable.
• **cm3** [8]: CModels3, a solver based on the definition of completion for disjunctive programs and the extension of loop formulas to the disjunctive case. CModels3 uses two SAT solvers in an interleaved way, the first for finding answer set candidates using the completion of the input program and loop formulas obtained during the computation, the second for verifying if the candidate model is indeed an answer set.
• **ssolve** [17]: is a search based native QBF solver that won the QBF Evaluation in 2004 on random (or probabilistic) benchmarks (performing very well also on non-random, or fixed, benchmarks), and performed globally (i.e., both on fixed and probabilistic benchmarks) well in the last two editions.

Note that we have not taken into account other solvers like Smodels$_{cc}$ [6] or Clasp [7] because our focus is on disjunctive ASP.

Benchmark Programs and Data. The proposed heuristic aims at improving the performance of DLV on disjunctive ASP programs. Therefore we focus on hard programs in this class, which is known to be able to express each problem of the complexity class Σ_2^P/Π_2^P. All of the instances that we have considered in our benchmark analysis have been derived from instances for 2QBF, the canonical problem for the second level of the polynomial hierarchy. This choice is motivated by the fact that many real-world, structured (i.e., fixed) instances in this complexity class are available for 2QBF on QBFLIB [18], and moreover, studies on the location of hard instances for randomly generated 2QBFs have been reported in [19–21].

The problem 2QBF is to decide whether a quantified Boolean formula (QBF) $\Phi = \forall X \exists Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = D_1 \wedge \ldots \wedge D_k$ is a CNF formula over $X \cup Y$, is valid.

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [22]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation requires $2 * (|X| + |Y|) + 1$ propositional predicates (with one dedicated predicate w), and consists of the following rules. Rules of the form $v \vee \bar{v}.$ for each variable $v \in X \cup Y$. Rules of the form $y \leftarrow w.$ $\bar{y} \leftarrow w.$ for each $y \in Y$. Rules of the form $w \leftarrow \bar{v}_1, \ldots, \bar{v}_m, v_{m+1}, \ldots, v_n.$ for each disjunction $v_1 \vee \ldots \vee v_m \vee \neg v_{m+1} \vee \ldots \vee \neg v_n$ in ϕ. The rule $\leftarrow \texttt{not}\ w$. The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has no answer set [22].

We have selected both random and structured QBF instances. The random 2QBF instances have been generated following recent phase transition results for QBFs [19–21]. In particular, the generation method described in [21] has been employed and the generation parameters have been chosen according to the experimental results reported in the same paper. First, we have generated 10 different sets of instances, each of which

[7] Note that all systems with h_{LB} heuristics exploit backjumping.

is labelled with an indication of the employed generation parameters. In particular, the label "A-E-C-ρ" indicates the set of instances in which each clause has A universally-quantified variables and E existentially-quantified variables randomly chosen from a set containing C variables, such that the ratio between universal and existential variables is ρ. For example, the instances in the set "3-3-70-0.8" are 6CNF formulas (each clause having exactly 3 universally-quantified variables and 3 existentially-quantified variables) whose variables are randomly chosen from a set of 70 containing 31 universal and 39 existential variables, respectively. In order to compare the performance of the systems in the vicinity of the phase transition, each set of generated formulas has an increasing ratio of clauses over existential variables (from 1 to max$r$). Following the results presented in [21], max$r$ has been set to 21 for each of the sets 3-3-70-*, and 12 for each of the 2-3-80-*. We have generated 10 instances for each ratio, thus obtaining, in total, 210 and 120 instances per set, respectively. Then, because such instances do not provide information about the scalability of the systems w.r.t. the total number of variables, we generated other sets. We took the "2-3-80-0.8" and "3-3-70-1.0" sets, we fixed the ratio of clauses over existential variables to the "harder" value for the DLV versions and vary the number of variables C (from 5 to max$C$, step 5), where max$C$ is 80 and 70, respectively. We have generated 10 instances for each point, thus obtaining, in total, 160 and 140 instances per set, respectively.

About the structured instances, we have analyzed:

- **Narizzano-Robot** - These are real-word instances encoding the robot navigation problems presented in [23], as used in the QBF Evaluation 2004 and 2005.
- **Ayari-MutexP** - These QBFs encode instances to problems related to the formal equivalence checking of partial implementations of circuits, as presented in [24].
- **Letz-Tree** - These instances consist of simple variable-independent subprograms generated according to the pattern: $\forall x_1 x_3 ... x_{n-1} \exists x_2 x_4 ... x_n (c_1 \wedge \ldots \wedge c_{n-2})$ where $c_i = x_i \vee x_{i+2} \vee x_{i+3}$, $c_{i+1} = \neg x_i \vee \neg x_{i+2} \vee \neg x_{i+3}$, $i = 1, 3, \ldots, n-3$.

The benchmark instances belonging to Letz-tree, Narizzano-Robot, Ayari-MutexP have been obtained from QBFLIB [18], including the 32 (resp. 40) Narizzano-Robot instances used in the QBF Evaluation 2004 (resp. 2005), and all the $\forall\exists$ instances from Letz-tree and Ayari-MutexP.

Results. All the experiments were performed on a 3GHz PentiumIV equipped with 1GB of RAM, 2MB of level 2 cache running Debian GNU/Linux. Time measurements have been done using the `time` command shipped with the system, counting total CPU time for the respective process.

We start with the results of the experiments with random 2QBF formulas. For every instance, we have allowed a maximum running time of 20 minutes. In Table 1 we report, for each system, the number of instances solved in each set within the time limit. Looking at the table, it is clear that the new look-back heuristic combined with the "mf" initialization (corresponding to the system dlv.mf) performed very well on these domains, being the version which was able to solve most instances in most settings, particularly on the 3-3-70-* sets. Also dlv.lf, in particular when combined with the "af" option, performed quite well, while the other variants do no seem to be very effective. Considering the look-ahead versions of DLV, dlv.ds performed reasonably well. Considering GnT and

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve
2-3-80-0.4	119	120	120	120	120	120	120	3	57	120
2-3-80-0.6	91	102	99	103	83	101	96	4	62	120
2-3-80-0.8	88	99	99	99	79	97	92	5	73	120
2-3-80-1.0	81	95	96	106	80	95	95	10	81	120
2-3-80-1.2	84	99	101	109	85	101	102	6	93	120
3-3-70-0.6	159	174	168	172	157	164	166	4	76	210
3-3-70-0.8	128	138	135	150	123	132	140	2	82	210
3-3-70-1.0	114	128	127	149	112	128	125	7	96	205
3-3-70-1.2	123	131	133	156	115	129	140	9	117	209
3-3-70-1.4	124	139	142	161	117	142	141	9	131	210
#Total	1111	1225	1220	1325	1071	1209	1217	59	868	1644

Table 1. Number of solved instances within timeout for Random 2QBF.

CModels3, we can note that they could solve few instances, while it is clear that Ssolve is very efficient, being able to solving almost all instances.

Figures 4 (resp. 5) show the results for the "2-3-80-0.8" (resp. "3-3-70-1.0") set, regarding scalability. For sake of readability, only the instances with an high number of variables are presented: GnT, Cmodels3, Ssolve and all the DLV versions solve all instances not reported. The left (resp. right) plot of each Figure contains the cumulative number of solved instances about all the DLV versions (resp. GnT, CModels3, Ssolve and the best version of DLV). Overall, on these particular sets, we can see that all the "look-back" versions of DLV scaled much better than GnT and CModels3, with very similar results among them (dlv.lf.af just solve one more instance (Fig. 5 left)). Ssolve managed to solve all instances, and in less time (not reported).

In Tables 2, 3 and 4, we report the results, in terms of execution time for finding one answer set, and/or number of instances solved within 20 minutes, about the groups: Narizzano-Robot, Ayari-MutexP and Letz-Tree, respectively. The last columns (AS?) indicate if the instance has an answer set (Y), or not (N), but for Table 2 where it indicates how many instances have answer sets. A "–" in these tables indicates a timeout. For h_{LB} heuristics, we experimented a few different values for "k", and we obtained the best results for k=100. However, it would be interesting to analyze more thoroughly the effect of the factor k.

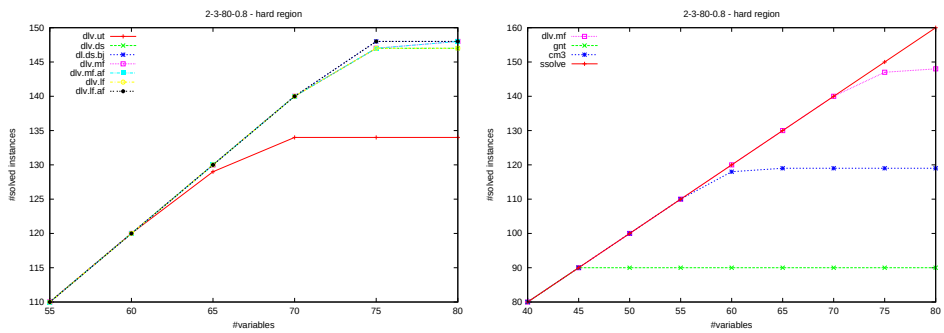


Fig. 4. Left: Number of solved instances by all DLV versions. Right: Number of solved instances by dlv.mf, GnT, CModels3 and Ssolve.

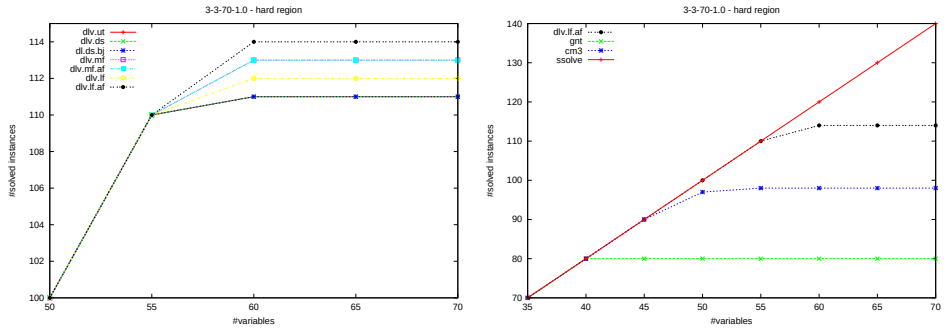


Fig. 5. Left: Number of solved instances by all DLV versions. Right: Number of solved instances by dlv.lf.af, GnT, CModels3 and Ssolve.

In Table 2 we report only the instances from the QBF Evaluation 2004 and 2005, respectively, which were solved within the time limit by at least one of the compared methods. In Table 2, dlv.mf was the only ASP and QBF solver able to solve all the reported 63 (23 for QBF Evaluation 2004 and 40 for QBF Evaluation 2005) instances, followed by Ssolve (60), CModels3 (58) and dlv.lf (47). Moreover, dlv.mf was always the fastest ASP system on each instance (sometimes dramatically, even if for lack of space we consider the instances on which it took more than 1 sec, and often faster than Ssolve, especially on the QBF Evaluation 2004 instances. On the QBF Evaluation 2005 instances, dlv.mf, Cmodels3 and Ssolve solved all of them, with a mean execution time of 228.07s, 189.74s and 76.91s, respectively. The "traditional" DLV versions could solve 10 instances, while dlv.ds.bj solved 21 instances, and took less execution time. This indicates the advantages of using a backjumping technique on these robot instances.

In Table 3, we then report the results for Ayari-MutexP. In that domain all the versions of DLV and Ssolve were able to solve all 7 instances, outperforming both CModels3 and GnT which solved only one instance. Comparing the execution times required by all the variants of dlv we note that, also in this case, dlv.mf is the best-performing version, while Ssolve scaled up much better. About the Letz-Tree domain, the DLV versions equipped with look-back heuristics solved a higher number of instances and required less CPU time (up to two orders of magnitude less) than all ASP competitors.

In particular, the look-ahead based versions of DLV, GnT and CModels3 could solve only 3 instances, while dlv.mf and dlv.lf solved 4 and 5 instances, respectively. Interestingly, here the "lf" variant is very effective in particular when combined with the "af" option, like in the random instances for testing scalability. It could solve the same number of instances as Ssolve, with Ssolve having better scaling capabilities.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
QBF Eval. 2004	10	10	11	23	12	15	12	5	18	20	5
QBF Eval. 2005	0	0	10	40	34	32	22	0	40	40	0
#Total	10	10	21	63	46	47	34	5	58	60	5

Table 2. Number of solved instances on Narizzano-Robot instances as selected in the QBF Evaluation 2004 and 2005. The last column indicates how many instances have answer sets.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
mutex-2-s	0.01	0.01	0.01	0.01	0.01	0.01	0.01	1.89	0.65	0.03	N
mutex-4-s	0.05	0.05	0.05	0.06	0.05	0.06	0.05	–	–	0.04	N
mutex-8-s	0.21	0.2	0.23	0.21	0.21	0.23	0.21	–	–	0.07	N
mutex-16-s	0.89	0.89	0.98	0.89	0.89	1.01	0.9	–	–	0.13	N
mutex-32-s	3.67	3.72	4.06	3.63	3.64	4.16	3.79	–	–	0.3	N
mutex-64-s	15.38	16.08	17.64	14.97	15.04	18.08	16.97	–	–	0.81	N
mutex-128-s	69.07	79.39	90.92	62.97	62.97	92.92	93.05	–	–	2.83	N
#Solved	7	7	7	7	7	7	7	1	1	7	

Table 3. Execution time (seconds) and number of solved instances on Ayari-MutexP instances.

Summarizing, dlv.ds.bj showed (especially on same sets of the random programs, and on the Narizzano-Robot instances) improvements w.r.t. the "traditional" DLV versions. Moreover, if equipped with look-back heuristics, DLV showed very positive performance, further strengthening the potential of look-back techniques. In all of the test cases presented, both random and structured, DLV equipped with look-back heuristics obtained good results both in terms of number of solved instances and execution time compared to traditionals DLV, GnT and CModels3. dlv.mf, the "classic" look-back heuristic, performed best in most cases, but good performance was obtained also by dlv.lf. The results of dlv.lf.af on the some random and Letz-Tree instances show that this option can be fruitfully exploited in some particular domains. We also included in the picture the QBF solver Ssolve: while often it showed very good results, on same domains, i.e., the Narizzano-Robot, dlv.mf performed better than Ssolve, both in terms of number of instances solved and CPU execution time. It should be also noted that the vast majority of the structured instances presented do not have answer sets, while the bigger advantages of dlv.mf over Ssolve on the Narizzano-Robot instances are obtained on the instances with answer sets.

5 Conclusions

We have described a general framework for employing look-back techniques in disjunctive ASP. In particular, we have designed a number of look-back based heuristics, addressing some key issues arising in this framework. We have implemented all proposed techniques in the DLV system, and carried out a broad experimental analysis on hard instances encoding 2QBFs, comprising both randomly generated instances and structured instances. It turned out that the proposed heuristics outperform the traditional (disjunctive) ASP systems DLV, GnT and CModels3 in most cases, and a rather simple approach

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	ssolve	AS?
exa10-10	0.18	0.17	0.17	0.04	0.1	0.06	0.06	0.12	0.03	0.01	N
exa10-15	7.49	7.09	7.31	0.34	0.71	0.48	0.38	6.46	0.73	0.01	N
exa10-20	278.01	264.53	275.1	12.31	17.24	5.43	2.86	325.26	67.56	0.02	N
exa10-25	–	–	–	303.67	432.32	44.13	19.15	–	–	0.02	N
exa10-30	–	–	–	–	–	166.93	129.54	–	–	0.05	N
#Solved	3	3	3	4	4	5	5	3	3	5	

Table 4. Execution time (seconds) and number of solved instances on Letz-Tree instances.

("dlv.mf") works particularly well, being performance-wise competitive with respect to "native" QBF solvers. A possible topic for future research is to further expand the range of look-back techniques in DLV by employing *learning* (the ability to record reasons in order to further avoid inconsistencies already encountered).

References

1. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proc. Fifth Intl Conference and Symposium, MIT Press (1988) 1070–1080
2. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
3. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
4. Prosser, P.: Hybrid Algorithms for the Constraint Satisfaction Problem. Computational Intelligence **9** (1993) 268–299
5. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001, ACM (2001) 530–535
6. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. LPNMR-7. LNCS 2923
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07),(2007) 386–392
8. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. LPNMR05. LNCS 3662
9. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using Database Optimization Techniques for Nonmonotonic Reasoning. In DDLP'99, Prolog Association of Japan (1999) 135–139
10. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications **19**(2) (2006) 155–172
11. Faber, W.: Enhancing Efficiency and Expressiveness in Answer Set Programming Systems. PhD thesis, TU Wien (2002)
12. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: IJCAI 2001, Seattle, WA, USA,(2001) 635–640
13. Faber, W., Ricca, F.: Solving Hard ASP Programs Efficiently. In: LPNMR'05. LNCS 3662
14. Faber, W., Leone, N., Ricca, F.: Solving Hard Problems for the Second Level of the Polynomial Hierarchy: Heuristics and Benchmarks. Intelligenza Artificiale **2**(3) (2005) 21–28
15. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL **7**(3) (2006) 499–562
16. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. LPNMR-7. LNCS 2923
17. Feldmann, R., Monien, B., Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In: AAAI(2000), AAAI Press (2000) 285–290
18. Narizzano, M., Tacchella, A.: QBF Solvers Evaluation page (2002) `http://www.qbflib.org/qbfeval/index.html/`.
19. Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: AI*IA 97. Italy, (1997) 207–218
20. Gent, I., Walsh, T.: The QSAT Phase Transition. In: AAAI. (1999)
21. Chen, H., Interian, Y.: A model for generating random quantified boolean formulas. In: Proceedings of IJCAI-05, Professional Book Center (2005) 66–71
22. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. AMAI **15**(3/4) (1995) 289–323
23. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. Artificial Intelligence **147**(1/2) (2003) 85–117
24. Ayari, A., Basin, D.A.: Bounded Model Construction for Monadic Second-Order Logics. In: Proc. of Computer Aided Verification, CAV 2000, Chicago, IL, USA, 15-19, 2000