Experimenting with Look-Back Heuristics for Hard ASP Programs*

Wolfgang Faber, Nicola Leone, Marco Maratea, and Francesco Ricca

Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
{faber,leone,maratea,ricca}@mat.unical.it

Abstract. Competitive native solvers for Answer Set Programming (ASP) perform a backtracking search by assuming the truth of literals. The choice of literals (the heuristic) is fundamental for the performance of these systems.
Most of the efficient ASP systems employ a heuristic based on look-ahead, that is, a literal is tentatively assumed and its heuristic value is based on its deterministic consequences. However, looking ahead is a costly operation, and indeed look-ahead often accounts for the majority of time taken by ASP solvers. For Satisfiability (SAT), a radically different approach, called look-back heuristic, proved to be quite successful: Instead of looking ahead, one uses information gathered during the computation performed so far, thus looking back. In this approach, atoms which have been frequently involved in inconsistencies are preferred.
In this paper, we carry over this approach to the framework of *disjunctive* ASP. We design a number of look-back heuristics exploiting peculiarities of ASP and implement them in the ASP system DLV. We compare their performance on a collection of hard ASP programs both structured and randomly generated. These experiments indicate that a very basic approach works well, outperforming all of the prominent disjunctive ASP systems — DLV (with its traditional heuristic), GnT, and CModels3 — on many of the instances considered.

1 Introduction

Answer set programming (ASP) is a comparatively novel programming paradigm, which has been proposed in the area of 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 [1]. The knowledge representation language of ASP is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, ASP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. Thus, ASP is strictly more powerful than SAT-based programming, as it allows for solving problems which cannot be translated to SAT in polynomial time (unless $P = NP$). For instance, several problems in diagnosis and planning under incomplete knowledge are complete for the complexity class Σ_2^P or Π_2^P [3, 4], and can be naturally encoded in ASP [5, 6].

* 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."

Since the model generators of native ASP systems are similar to the DPLL procedure, employed in many SAT solvers, the heuristic (branching rule) for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) is fundamentally important for the efficiency of an ASP system. Many of the efficient ASP systems, and especially the disjunctive ASP systems, employ a heuristic based on look-ahead. This means that the available choices are hypothetically assumed, their deterministically entailed consequences are computed, and a heuristic function is evaluated on the result. The look-ahead approach has been shown to be effective [9, 10] and it bears the additional benefit of detecting choices that deterministically cause an inconsistency. However, the sheer number of potential choices and the costly computations done for each of these makes the look-ahead a rather costly operation. Indeed, look-ahead often accounts for the majority of time taken by ASP solvers.

In SAT, a radically different approach, called look-back heuristics, proved to be quite successful [11]: Instead of making tentative assumptions and thus trying to look into the future of the computation, one uses information already collected during the computation so far, thus looking back; atoms which have been most frequently involved in inconsistencies are heuristically preferred (following the intuition that "most constrained" atoms are to be decided first).

In this paper, we take this approach from SAT to the framework of disjunctive ASP, trying to maximally exploit peculiarities of ASP, and experiment with alternative ways of addressing the key issues arising in this framework. The main contributions of the paper are as follows.

- We define a framework for look-back heuristics in disjunctive ASP. We build upon the work in [12], which describes a calculus identifying reasons for encountered inconsistencies in order to allow backjumping (i.e., avoiding backtracking to choices which do not contribute to an encountered inconsistency). For obtaining a "most constrained choices first" strategy, we prefer those choices that were the reasons for earlier inconsistencies. Our framework exploits the peculiarities of disjunctive ASP, a relevant feature concerns the full exploitation of "hidden" inconsistencies which are due to the failure of stable-model checks.
- We design a number of look-back heuristics for disjunctive ASP. In particular, we study different ways of making choices when information on inconsistencies is poor (e.g., at the beginning of the computation, when there is still nothing to look back to). We consider also different ways of choosing the "polarity" (positive or negative) of the atoms to be taken (intuitively negative choices keep the interpretation closer to minimality, which is mandatory in ASP).
- We implement all proposed heuristics in the ASP system DLV [7].
- We carry out an experimental evaluation of all proposed heuristics on programs encoding random and structured 2QBF formulas, the prototypical problem for Π_2^P (the class characterizing hard disjunctive ASP programs).

The results are very encouraging, the new heuristics perform very well compared to the traditional disjunctive ASP systems DLV, GnT [8] and CModels3 [13]. In particular, a very basic heuristic outperforms all other systems on a large part of the considered instances.

To our knowledge, while look-back heuristics have been widely studied for SAT (see, e.g., [11] [14], [15]), so far only few works have studied look-back heuristics for $\vee$-free ASP [16, 17], and this is the first paper on look-back heuristics for disjunctive ASP.[1]

2 Answer Set Programming Language

A *(disjunctive) rule* r is a formula

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

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 $B_\mathcal{P}$.[2] 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 $\mathcal{P}$.

Example 1. For the positive program $\mathcal{P}_1 = \{a \vee b \vee c.\ ,\ \coloneq a.\}$, $\{b, \texttt{not } a, \texttt{not } c\}$ and $\{c, \texttt{not } a, \texttt{not } b\}$ are the only answer sets.

For the positive program $\mathcal{P}_2 = \{a \vee b \vee c.\ ,\ \coloneq a.\ ,\ b \coloneq c.\ ,\ c \coloneq b.\}$, $\{b, c, \texttt{not } a\}$ is the only answer set. ∎

[1] The disjunctive ASP system CModels3 [13] "indirectly" uses look-back heuristics, since it works on top SAT solvers which may employ this technique.

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

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 $Ground(\mathcal{P})^X$.

Example 2. Given the (general) program $\mathcal{P}_3 = \{a \vee b \text{:-} c.,\ b \text{:-} \texttt{not}\ a, \texttt{not}\ c.,\ a \vee c \text{:-} \texttt{not}\ b.\}$ and $I = \{b, \texttt{not}\ a, \texttt{not}\ c\}$, the reduct $\mathcal{P}_3^I$ is $\{a \vee b \text{:-} c., b.\}$. It is easy to see that I is an answer set of $\mathcal{P}_3^I$, and for this reason it is also an answer set of $\mathcal{P}_3$. ∎

3 Answer Set Computation

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.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[3] The subsequent computations are then performed on $ground(\mathcal{P})$.

```
Function ModelGenerator(I: Interpretation): Boolean;
var   inconsistency: Boolean;
begin
    I := DetCons(I);
    if I = 𝓛 then return False; (* inconsistency *)
    if no atom is undefined in I then return IsAnswerSet(I);
    Select an undefined ground atom A according to a heuristic;
    if ModelGenerator(I ∪ {A}) then return True;
    else return ModelGenerator(I ∪ {not A});
end;
```

Fig. 1. Computation of Answer Sets

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. The ModelGenerator function is initially called with parameter I set to the empty interpretation.[4] 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

[3] 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 $\mathcal{P}$.

[4] 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.

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 next section, we 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.

4 Reasons for Literals

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 $a \text{:-} b, c, \texttt{not}\ \ d.$, 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 [12] 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 different and 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 Section 3 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). Note that unsatisfied rules with unfounded atoms in their heads are not relevant for stability and hence do not contribute to the reason. The information on reasons for inconsistencies can be exploited for backjumping, as described in [12], by going back to the closest choice which is a reason for the inconsistency, rather than always to the immediately preceding choice. In the remainder of this paper, we will describe extensions of a backjumping-based solver by further exploiting the information provided by reasons. In particular, in the following section we describe how reasons for inconsistencies can be exploited for defining a look-back heuristic.

5 Heuristics

In this section we will first describe the two main heuristics for DLV (based on look-ahead), and subsequently define several new heuristics based on reasons, which are computed as side-effects of the backjumping technique. Throughout this section, 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} [9], which has recently been refined to yield the heuristic h_{DS} [18], 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 [9], 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 same criteria involving UT atoms.

Enhanced Heuristic of DLV **(h_{DS}).**

The heuristic h_{DS}, proposed in [19] 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 (as discussed in Section 4) 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.

Note also that the complexity of look-ahead heuristics is in general quadratic (in the number of atoms), and becomes linear if a bound on the number of atoms to be analyzed is a-priori known. On the other hand, h_{LB} heuristics are constant time, but need the values $V(L)$ to be re-ordered after having chosen k literals.

6 Experiments

We have implemented all the proposed heuristics in DLV; in this section, we report on their experimental evaluation.

6.1 Compared Methods

For our experiments, we have compared several versions of DLV [7], 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. 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}.[5]
• **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** [8]: 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** [13]: 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.

Note that we have not taken into account other solvers like Smodels$_{cc}$ [16] or Clasp [17] because our focus is on disjunctive ASP.

6.2 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. All of the instances that we have considered in our benchmark analysis have been derived from instances for 2QBF, the canonical Σ_2^P-complete problem. This choice is motivated by the fact that many real-world, structured instances for problems in Σ_2^P are available for 2QBF on QBFLIB [20], and moreover, studies on the location of hard instances for randomly generated 2QBFs have been reported in [21–23].

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.

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

The transformation from 2QBF to disjunctive logic programming is a slightly altered form of a reduction used in [24]. 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 $\leftarrow$ `not` w. The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ has no answer set [24].

We have selected both random and structured QBF instances. The random 2QBF instances have been generated following recent phase transition results for QBFs [21–23]. In particular, the generation method described in [23] has been employed and the generation parameters have been chosen according to the experimental results reported in the same paper. We have generated 13 different sets of instances, each of which 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 "3-3-50-0.8" 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 50 containing 22 universal and 28 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 [23], max$r$ has been set to 21 for each of the sets 3-3-50-* and 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.

The structured instances we have analyzed are:

- **Narizzano-Robot** - These are real-word instances encoding the robot navigation problems presented in [25].
- **Ayari-MutexP** - These QBFs encode instances to problems related to the formal equivalence checking of partial implementations of circuits, as presented in [26].
- **Letz-Tree** - These instances consist of simple variable-independent subprograms generated according to the pattern: $\forall x_1 x_3 \ldots x_{n-1} \exists x_2 x_4 \ldots x_n (c_1 \wedge \ldots \wedge c_{n-2})$ 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 [20], including the 32 Narizzano-robot instances used in the QBF Evaluation 2004, and all the $\forall\exists$ instances from Letz-tree and Ayari-MutexP.

6.3 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.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3
2-3-80-0.4	106	114	114	107	100	109	103	3	47
2-3-80-0.6	83	88	89	92	71	90	83	4	58
2-3-80-0.8	78	92	95	93	70	89	86	3	65
2-3-80-1.0	78	90	91	98	66	88	85	8	77
2-3-80-1.2	72	89	94	105	74	93	95	4	87
3-3-50-0.8	210	210	210	210	210	210	210	21	166
3-3-50-1.0	191	205	202	201	199	203	202	30	163
3-3-50-1.2	196	207	206	208	203	207	206	41	191
3-3-70-0.6	126	136	135	140	127	131	131	1	61
3-3-70-0.8	112	115	115	128	103	113	119	0	68
3-3-70-1.0	91	108	109	137	94	110	108	3	82
3-3-70-1.2	104	121	122	139	90	117	121	5	108
3-3-70-1.4	106	123	124	151	98	131	126	3	118
#Total	1552	1698	1706	1809	1505	1691	1675	126	1291

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

We start with the results of the experiments with random 2QBF formulas. For every instance, we have allowed a maximum running time of 6 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 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 CModels3, we can note that they could solve comparatively few instances.

Comparing between the 3-3-50-* and 3-3-70-* settings, we can see that dlv.mf is the system that scales best: It is on the average when considering 50 variables, while it is considerably better when considering 70 variables.

We do not report details on the execution times due to lack of space, as aggregated results such as average or median are problematic because of the many timeouts. However, for 3-3-50-0.8 all DLV-based systems terminated, and here the average times do not differ dramatically, the best being dlv.ds (23.62s), dlv.mf (25.26s) and dlv.ds.bj (26.02s). In other settings, such as 2-3-80-0.6, we observe that dlv.mf is the best on average time over the solved instances (18.31s), while all others solve fewer instances with a higher average time. Similar considerations hold for 3-3-70-1.2 where dlv.mf solves 17 instances more than the second best, dlv.ds.bj, yet its average time is about 30% lower (22.93s vs. 34.89s).

In Tables 2, 3 and 4, we report the results, in terms of execution time for finding one answer set, and number of instances solved within 11 minutes, about the groups: Letz-Tree, Narizzano-Robot, and Ayari-MutexP, respectively. The last columns (AS?) indicate if the instance has an answer set (Y), or not (N). 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. In Table 2 we report only the instances which were solved within the time limit by at least one of the compared methods. On these instances, dlv.mf was able to solve all the shown 23 instances, followed by CModels3 (18) and dlv.lf (15). Moreover, dlv.mf was also always the fastest system on each instance (sometimes dramatically), if we consider the instances on which it took more than 1 sec.

	dlv.ut	dlv.ds	dlv.ds.bj	dlv.mf	dlv.mf.af	dlv.lf	dlv.lf.af	gnt	cm3	AS?
2-38.1	0.31	0.31	0.31	0.36	0.39	0.37	0.33	44.4	1.31	N
2-64.1	0.30	0.30	0.32	0.36	0.39	0.37	0.33	43.77	1.3	N
2-93.1	0.31	0.30	0.32	0.36	0.39	0.37	0.33	44.35	1.3	N
2-69.4	–	–	–	536.97	–	–	–	–	263.05	N
2-3.5	–	–	–	14.39	352.94	387.11	364.89	–	431.55	N
2-61.6	–	–	–	629.35	–	–	–	–	–	Y
2-72.7	–	–	–	14.21	–	–	–	–	390.62	N
3-17.2	14.26	7.45	5.67	4.59	5.85	8.22	7.31	–	8.1	N
3-62.4	–	–	–	362.57	–	–	–	–	211.68	N
3-80.4	–	–	–	404.93	–	–	–	–	239.96	N
4-78.1	0.30	0.30	0.31	0.43	0.51	0.37	0.33	37.3	1.31	N
4-21.2	13.40	6.84	5.27	3.60	4.25	5.68	7.31	–	8.14	N
4-73.2	13.36	6.80	4.07	2.49	3.18	4.72	6.65	–	6.68	N
4-91.4	–	–	–	236.41	–	–	–	–	212.76	N
4-85.5	–	–	504.61	3.59	156.60	109.04	372.78	–	103.04	N
4-87.8	–	–	–	244.47	–	600.36	–	–	–	Y
5-29.1	0.30	0.30	0.31	0.43	0.51	0.36	0.32	37.1	1.3	N
5-5.2	13.39	6.83	4.09	2.50	3.18	4.73	6.68	–	6.66	N
5-75.3	655.78	188.80	71.56	14.70	31.44	62.93	47.85	–	34.74	N
5-18.5	–	–	–	357.04	–	–	–	–	–	Y
5-59.5	–	–	–	357.15	–	–	–	–	–	Y
5-55.6	–	–	–	5.51	–	233.23	–	–	219.39	N
5-4.9	–	–	–	89.16	–	–	–	–	–	Y
#Solved	10	10	11	23	12	15	12	5	18	

Table 2. Execution time (seconds) and number of solved instances on Narizzano-Robot instances.

In Table 3, we then report the results for Ayari-MutexP. In that domain all the versions of DLV 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.

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 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.

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

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

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

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

Summarizing, DLV equipped with look-back heuristics showed very positive performance in all of the test cases presented, both random and structured, obtaining 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 Letz-Tree instances show that this option can be fruitfully exploited in some particular domains.

7 Conclusions

We have defined a general framework for employing look-back heuristics in disjunctive ASP, exploiting the peculiar features of this setting. We have designed a number of look-back based heuristics, addressing some key issues arising in this framework. We have implemented all proposed heuristics in the DLV system, and carried out experiments on hard instances encoding 2QBFs, comprising randomly generated instances, generated according to the method proposed in [23], and structured instances from the QBFLIB archive (Letz-Tree, Narizzano-Robot, Ayari-MutexP). 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.mf") works particularly well.

References

1. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proceedings of the 16th International Conference on Logic Programming (ICLP'99), Las Cruces, New Mexico, USA, The

MIT Press (1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22**(3) (1997) 364–418
3. Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In Dean, T., ed.: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999, Stockholm, Sweden, Morgan Kaufmann Publishers (1999) 1192–1197
4. Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. Journal of the ACM **42**(1) (1995) 3–42
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
6. Leone, N., Rosati, R., Scarcello, F.: Enhancing Answer Set Planning. In Cimatti, A., Geffner, H., Giunchiglia, E., Rintanen, J., eds.: IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. (2001) 33–42
7. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
8. Janhunen, T., Niemelä, I.: Gnt - a solver for disjunctive logic programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of LNAI., Springer (2004) 331–335
9. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, Seattle, WA, USA, Morgan Kaufmann Publishers (2001) 635–640
10. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
11. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001, ACM (2001) 530–535
12. Ricca, F., Faber, W., Leone, N.: A Backjumping Technique for Disjunctive Logic Programming. AI Communications – The European Journal on Artificial Intelligence **19**(2) (2006) 155–172
13. Lierler, Y.: Disjunctive Answer Set Programming via Satisfiability. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 447–451
14. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD 2001). (2001) 279–285
15. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver. In: Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France, IEEE Computer Society (2002) 142–149
16. Ward, J., Schlipf, J.S.: Answer Set Programming with Clause Learning. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the 7th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR-7). Volume 2923 of LNAI., Springer (2004) 302–313
17. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07). (2007) To appear.
18. Faber, W., Ricca, F.: Solving Hard ASP Programs Efficiently. In Baral, C., Greco, G., Leone, N., Terracina, G., eds.: Logic Programming and Nonmonotonic Reasoning — 8th In-

ternational Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings. Volume 3662 of Lecture Notes in Computer Science., Springer Verlag (2005) 240–252
19. 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
20. Narizzano, M., Tacchella, A.: QBF Solvers Evaluation page (2002) `http://www.qbflib.org/qbfeval/index.html/`.
21. Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: Proceedings of the 5th Congress: Advances in Artificial Intelligence of the Italian Association for Artificial Intelligence, AI*IA 97. Lecture Notes in Computer Science, Rome, Italy, Springer Verlag (1997) 207–218
22. Gent, I., Walsh, T.: The QSAT Phase Transition. In: Proceedings of the 16th AAAI. (1999)
23. Chen, H., Interian, Y.: A model for generating random quantified boolean formulas. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05), Professional Book Center (2005) 66–71
24. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. Annals of Mathematics and Artificial Intelligence **15**(3/4) (1995) 289–323
25. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. Artificial Intelligence **147**(1/2) (2003) 85–117
26. Ayari, A., Basin, D.A.: Bounded Model Construction for Monadic Second-Order Logics. In: Prooceedings of Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Chicago, IL, USA (2000)