Ann Math Artif Intell (2007) 51:229–266
DOI 10.1007/s10472-008-9087-4

On look-ahead heuristics in disjunctive logic programming

Wolfgang Faber · Nicola Leone · Gerald Pfeifer · Francesco Ricca

Published online: 5 March 2008

Abstract Disjunctive logic programming (DLP), also called answer set programming (ASP), is a convenient programming paradigm which allows for solving problems in a simple and highly declarative way. The language of DLP is very expressive and able to represent even problems of high complexity (every problem in the complexity class $\Sigma_2^P = \mathrm{NP}^{\mathrm{NP}}$). During the last decade, efficient systems supporting DLP have become available. Virtually all of these systems internally rely on variants of the Davis–Putnam procedure (for deciding propositional satisfiability [SAT]), combined with a suitable model checker. The heuristic 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) dramatically affects the performance of a DLP system. While heuristics for SAT have received a fair deal of research, only little work on heuristics for DLP has been done so far. In this paper, we design, implement, optimize, and experiment with a number of heuristics for DLP. We focus on different look-ahead heuristics, also called "dynamic heuristics" (the DLP equivalent of unit propagation [UP] heuristics for SAT). These are branching rules where the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. We motivate and formally define a number of look-ahead heuristics for DLP programs. Furthermore, since look-ahead heuristics are computationally expensive, we design

Some results in this paper have been presented in preliminary form at IJCAI-01 [16], LPNMR'01 [17], and LPNMR'05 [18].

W. Faber · N. Leone (✉) · G. Pfeifer · F. Ricca
Department of Mathematics, University of Calabria, 87036 Rende (CS), Italy
e-mail: leone@mat.unical.it

W. Faber
e-mail: faber@mat.unical.it

G. Pfeifer
e-mail: gerald@mat.unical.it

F. Ricca
e-mail: ricca@mat.unical.it

two techniques for optimizing the burden of their computation. We implement all the proposed heuristics and optimization techniques in DLV—the state-of-the-art implementation of disjunctive logic programming, and we carry out experiments, thoroughly comparing the heuristics and optimization techniques on a large number of instances of well-known benchmark problems. The results of these experiments are very interesting, showing that the proposed techniques significantly improve the performance of the DLV system.

Keywords Artificial intelligence · Logic programming · Nonmonotonic reasoning · Answer set programming · Heuristics · Stable models · Efficient evaluation

Mathematics Subject Classifications (2000) 68N17 · 68T27 · 68T20

1 Introduction

Disjunctive logic programming (DLP), first proposed by Jack Minker in the early eighties [34], is a powerful language for knowledge representation and reasoning. Its semantics, nowadays usually based on the notion of stable models [21] to account for negation,[1] is fully declarative. Disjunctive logic programming is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, DLP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [11]. Thus, DLP is strictly more powerful than both OR-free ASP and SAT-based programming, as it allows us to solve problems which cannot be translated to SAT in polynomial time (unless $P = NP$). The high expressive power of DLP can be profitably exploited in AI, which often has to deal with problems of high complexity. For instance, problems in diagnosis and planning under incomplete knowledge are complete for the complexity class Σ_2^P or Π_2^P [10, 40], and can be naturally encoded in DLP [1, 26].

The high expressiveness of disjunctive logic programming comes at the price of a high computational cost in the worst case, which strongly motivates the research in the area of heuristics and optimization techniques for the efficient DLP implementation.

After the instantiation process, where variables are eliminated and a ground program is generated, the core of the second phase of the computation of a DLP system is model generation, where a model of the program is produced, which is then subjected to a stability check [25]. For the generation of models, DLP systems employ procedures which are similar to Davis–Putnam procedures used in SAT solvers. As for 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 a model generation procedure, and dramatically affects the overall performance of a DLP system. While a lot of work has been done in AI developing new heuristics and comparing alternative heuristics for SAT (see, e.g., [7, 19, 23, 24, 28–30, 36]), only

[1]Stable models are also called answer sets, and disjunctive logic programming is often referred to as answer set programming (ASP).

little work has been done so far for DLP systems. In particular, we are not aware of any previous comparison between heuristics for DLP.

In this paper, we evaluate different heuristics for DLP systems. In particular, we consider DLV [25]—the state-of-the-art DLP system—and focus on different look-ahead heuristics, also called "dynamic heuristics" (the DLP equivalent of UP heuristics for SAT), that is, branching rules where the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences, and possibly combining it with the result of taking Q false (and computing the consequences) as well. Look-ahead heuristics, employed in competitive and well-assessed ASP systems like DLV [25] and Smodels [43], have been shown to be effective and they bear the additional benefit of detecting choices that deterministically cause an inconsistency, thereby pruning the search space. However, looking ahead is a comparatively costly operation, the computation of this kind of heuristics can be very expensive, since the number of literals to be "looked-ahead" may be very large, and indeed look-ahead often accounts for the majority of time taken by DLP solvers. To tackle this problem, we define two techniques for optimizing the computation of look-ahead heuristics, and discuss their properties. Moreover, we carry out an ample experimental activity comparing the different heuristics and the proposed techniques.

In sum, the contribution of the paper is the following:

- ▶ We formally define a number of look-ahead heuristics for DLP programs. In particular, some heuristics come from the DLP adaptation of heuristics for SAT and non-disjunctive ASP systems, and some heuristics are defined "ad hoc" to efficiently deal with Σ_2^P/Π_2^P-hard DLP programs.
- ▶ We define two techniques for optimizing the computation of look-ahead heuristics. One of these is based on the demonstration of an equivalence theorem, allowing us to recognize pairs of literals which are "a priori" guaranteed to have precisely the same heuristic value (this technique allows for avoiding 50% of the look-aheads in several cases including, e.g., Hamiltonian Path and 3SAT programs). The other technique is based on a two-layered approach. A computationally cheap heuristic criterion singles out the (sub)set of atoms to be considered for the heuristic involving look-ahead, which is applied only on the atoms in this set.
- ▶ We implement all the proposed heuristics and optimization techniques in DLV—the state-of-the-art implementation of disjunctive logic programming.
- ▶ We carry out an experimental activity comparing the heuristics and the optimization techniques on a large number of instances of well-known benchmark problems. In particular, we consider also two benchmark problems which are hard for the second level of the polynomial hierarchy, and strictly require the full power of disjunction.

The experiments show interesting results: a newly defined heuristic (h_4) on average outperforms the other heuristics on the set of considered benchmarks. Moreover, the two proposed optimization techniques are effective, they both reduce the computational cost of h_4, and can be profitably combined. These results led us to incorporate the heuristic and the optimization techniques in DLV: the current release of DLV adopts h_4 and both optimization techniques by default.

The organization of the paper is as follows. In Section 2 the syntax and semantics of DLP is reviewed, and some of its properties are recalled. In Section 3, the computational core of DLV is presented and the techniques used for evaluating heuristic functions are detailed. Then, in Section 4 a number of heuristic criteria are described, and in Section 5 two optimization techniques, that reduce the amount of time needed to evaluate the heuristics, are introduced. In Section 6 we report on the benchmarks performed to assess the impact of both heuristics and optimization techniques. Eventually, in Section 7 we draw our conclusions and discuss related work.

2 Language

In this section, we provide a brief introduction to the syntax and semantics of disjunctive logic programming (DLP). For further background see [8, 21, 35].

2.1 Syntax

A *disjunctive rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms[2], $n \geq 0$, $m \geq k \geq 0$, and $n+m \geq 1$. A literal is either an atom a (in this case, it is *positive*) or its default negation $\texttt{not}\ a$ (in this case, it is *negative*). Given a rule r, let $H(r) = \{a_1, ..., a_n\}$ denote the set of head literals, $B^+(r) = \{b_1, ..., b_k\}$ and $B^-(r) = \{\texttt{not}\ b_{k+1}, ..., \texttt{not}\ b_m\}$ the sets of positive and negative body literals, and $B(r) = B^+(r) \cup B^-(r)$ the set of body literals.

Given a literal l, we define its *complementary literal* $\texttt{not}.l$ as follows: $\texttt{not}.l = a$ if l is of the form $\texttt{not}\ a$, otherwise $\texttt{not}.l = \texttt{not}\ l$. Given a set L of literals, $\texttt{not}.L = \{\texttt{not}.l \mid l \in L\}$.

A rule r with $B^-(r) = \emptyset$ is called *positive*. A rule with $H(r) = \emptyset$ is referred to as *integrity constraint* or just *constraint*. If the body $B(r)$ is empty we usually omit the :– sign.

A *disjunctive logic program* $\mathcal{P}$ is a finite set of rules; $\mathcal{P}$ is a *positive* program if all rules in $\mathcal{P}$ are positive (i.e., not-free). An object (atom, rule, etc.) containing no variables is called *ground* or *propositional*. A rule is *safe* if each variable in that rule also appears in at least one positive literal in the body of that rule. A program is safe if each of its rules is safe and in the following we will assume that all programs are safe.

Example 1 Consider the following program:

$$\begin{aligned}
&r_1{:}\ a(X) \ \vee \ b(X) \ {:}{-}\ c(X, Y), d(Y), \texttt{not}\ e(X).\\
&r_2{:}\ {:}{-}\ c(X, Y), k(Y), e(X), \texttt{not}\ b(X)\\
&r_3{:}\ m \ {:}{-}\ n, o, a(1).\\
&r_4{:}\ c(1, 2).
\end{aligned}$$

[2]For simplicity, we do not consider strong negation in this paper. It can be emulated by introducing new atoms and integrity constraints.

r_1 is a disjunctive rule with $H(r_1) = \{a(X), b(X)\}$, $B^+(r_1) = \{c(X, Y), d(Y)\}$, and $B^-(r_1) = \{\texttt{not}\ e(X)\}$. r_2 is an integrity constraint with $B^+(r_2) = \{c(X, Y), k(Y), e(X)\}$, and $B^-(r_2) = \{\texttt{not}\ b(X)\}$. r_3 is a ground, positive, and non-disjunctive rule with $H(r_3) = \{m\}$, $B^+(r_3) = \{n, o, a(1)\}$, and $B^-(r_3) = \emptyset$. r_4, finally, is a fact (note that :– is omitted). Moreover, all of the rules are safe.

2.2 Semantics

The semantics of a disjunctive logic program is given by its stable models [21, 39], which we briefly review in this section.

Herbrand Universe 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 $U_{\mathcal{P}}$.

Ground Instantiation 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* of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

Interpretation 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, viz. $L \cap \texttt{not}.L = \emptyset$. 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. An interpretation I is *total* if, for each atom a in $B_{\mathcal{P}}$, either a or $\texttt{not}.a$ is in I, viz. $I \cup \texttt{not}.I = B_{\mathcal{P}}$, or in other words, if no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I.

Stable Models For every program, we define its stable models using its ground instantiation in two steps: First we define the stable models of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define stable models of general programs. In the following, we will assume for simplicity that $\mathcal{P}$ already denotes the ground instantiation of a given program.

Let r be a ground rule in $\mathcal{P}$. The head of r is *true* w.r.t. I if there is $a \in H(r)$ such that a is true w.r.t. I (i.e., some atom in $H(r)$ is true w.r.t. I). The body of r is *true* w.r.t. I if $\forall \ell \in B(r)$, ℓ is true w.r.t. I (i.e. all literals in $B(r)$ are true w.r.t. I). The body of r is *false* w.r.t. I if $\exists \ell \in B(r)$ such that ℓ is false w.r.t. I (i.e., some literal in $B(r)$ is false w.r.t. I). The rule r is *satisfied* (or *true*) w.r.t. I if its head is true w.r.t. I or its body is false w.r.t. I.

A total interpretation M is a *model* for $\mathcal{P}$ if for every $r \in \mathcal{P}$ 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 a *stable model* for a positive program $\mathcal{P}$ if its positive part is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

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

Example 2 Consider the positive programs:

$$p_1 = \{a \vee b \vee c.\,,\ :\!-a.\}$$

$$\mathcal{P}_2 = \{a \vee b \vee c.\,,\ :\!-a.\,,\ b\!:\!-c.\,,\ c\!:\!-b.\}$$

The stable models of $\mathcal{P}_1$ are $\{\texttt{not}\ a, b, \texttt{not}\ c\}$ and $\{\texttt{not}\ a, \texttt{not}\ b, c\}$, while $\{\texttt{not}\ a, b, c\}$ is the only stable model of $\mathcal{P}_2$.

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 (1) deleting all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X and (2) deleting the negative body from the remaining rules.

A stable model of a general ground program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is a stable model of $\mathcal{P}^X$.

Example 3 Given the following program

$$\mathcal{P}_3 = \{a \vee b\ :-c.,\ b\ :-\texttt{not}\ a, \texttt{not}\ c.,\ a \vee c-:\texttt{not}\ b.\}$$

and the interpretation $I = \{\texttt{not}\ a, b, \texttt{not}\ c\}$. The reduct $\mathcal{P}_3^I$ is $\{a \vee b\ :-\ c., b.\}$. I is a stable model of $\mathcal{P}_3^I$, and for this reason it is also a stable model of $\mathcal{P}_3$. Now consider the interpretation $J = \{a, \texttt{not}\ b, \texttt{not}\ c\}$. The reduct $\mathcal{P}_3^J$ is $\{a \vee b\ :-\ c.\,,\ a \vee c.\}$. It can be easily verified that J is a stable model of $\mathcal{P}_3^J$, and thus also a stable model of $\mathcal{P}_3$. On the other hand, for $K = \{\texttt{not}\ a, \texttt{not}\ b, c\}$ the reduct $\mathcal{P}_3^K$ is equal to $\mathcal{P}_3^J$, and it is easy to see that K is not a stable model of $\mathcal{P}_3^K$. Therefore K is also not a stable model of $\mathcal{P}_3$.

2.3 Some DLP properties

In this section, we recall some properties of (ground) disjunctive logic programs.

Supportedness Given an interpretation I for a ground program $\mathcal{P}$, we say that a ground atom A is *supported* in I if there exists a *supporting* rule r in the ground instantiation of $\mathcal{P}$ such that the body of r is true w.r.t. I and A is the only true atom in the head of r.

Proposition 1 [2, 27, 33] *If M is a stable model of a program $\mathcal{P}$, then all atoms in M are supported.*

Head-Cycle Free Programs Another relevant property of disjunctive logic programs is head-cycle freeness (HCF). With every ground program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (N, E)$, called the *dependency graph* of $\mathcal{P}$, where (1) the nodes in N are the atoms of $\mathcal{P}$, (2) there is an arc in E from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b appears in the head of r and a appears in the positive body of r.

The graph $DG_{\mathcal{P}}$ singles out the dependencies of the head atoms of a rule r from the positive atoms in its body. Negative literals cause no arc in $DG_{\mathcal{P}}$.

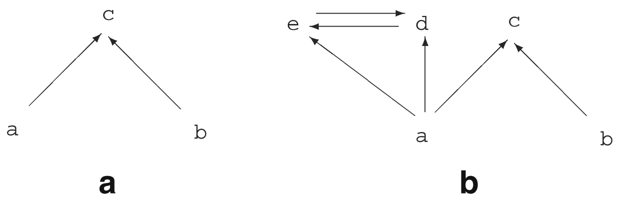


Fig. 1 Graphs (**a**) $DG_{\mathcal{P}_4}$, and (**b**) $DG_{\mathcal{P}_5}$

Example 4 Consider the following two programs:

$$\mathcal{P}_4 = \{a \vee b\,.\,,\; c{:}{-}a.\,,\; c{:}{-}b\,.\}$$

$$\mathcal{P}_5 = \mathcal{P}_4 \cup \{d \vee e{:}{-}a.\,,\; d{:}{-}e.\,,\; e{:}{-}d, \texttt{not}\; b\,.\}$$

The dependency graph $DG_{\mathcal{P}_4}$ is depicted in Fig. 1a, while the dependency graph $DG_{\mathcal{P}_5}$ is depicted in Fig. 1b.

The dependency graphs allow us to define HCF programs [3]. A program $\mathcal{P}$ is *HCF* iff there is no rule r in $\mathcal{P}$ such that two atoms occurring in the head of r occur in a single cycle of $DG_{\mathcal{P}}$.

Example 5 The dependency graphs given in Fig. 1 reveal that program $\mathcal{P}_4$ of Example 4 is HCF and that program $\mathcal{P}_5$ is not HCF, as rule $d \vee e{:}{-}a$ contains in its head two atoms belonging to the same cycle of $DG_{\mathcal{P}_5}$.

It has been shown that HCF programs are computationally easier than general (non-HCF) programs.

Proposition 2 [3, 11] *Deciding whether an atom belongs to some stable model of a ground HCF program $\mathcal{P}$ is* NP*-complete, while deciding whether an atom belongs to some stable model of a ground (non-HCF) program $\mathcal{P}$ is Σ_2^P-complete.*

A *component* C of a dependency graph DG is a maximal subgraph of DG such that each node in C is reachable from any other node in C. The *subprogram* of C consists of all rules having some atom from C in the head. An *atom* is non-HCF if the subprogram of its component is non-HCF.

3 Computational framework

In this section, we describe the main steps of the computational process performed by DLP systems. We will refer particularly to the computational engine of the DLV system [12, 25, 27], which will be used for the experiments, but also other DLP systems like Smodels [43] and Clasp [20] employ a very similar procedure.[4]

In general, a logic program $\mathcal{P}$ contains variables. The first step of a computation of a DLP system, performed by the so-called instantiator procedure (or grounding),

[4]Other solvers like Cmodels [31] and ASSAT [32] use a different architecture based on transformations to SAT.

eliminates these variables and generates a ground instantiation of $\mathcal{P}$. This process has been optimized in the DLV system where a (usually much smaller) subset of all syntactically constructible instances of the rules of $\mathcal{P}$, that has precisely the same stable models as $\mathcal{P}$ [14] is computed. The nondeterministic part of the computation is then performed on this ground program generated by the instantiator. For brevity, in the sequel $\mathcal{P}$ shall refer to the simplified ground program.

3.1 Model generation procedure

The heart of the computation is performed by the model generator, which is sketched in Fig. 2. Note that for reasons of presentation, the description here is quite simplified. For instance, the actual implementation computes and outputs all or a fixed number of stable models rather than just deciding whether stable models exist; however, the extension is quite straightforward and not important for this paper. Moreover, the implementation uses quite advanced data structures, which are also not important for the discussion of this paper. A more detailed description can be found in [12].

Roughly, the model generator produces some "candidate" stable models. Each candidate I is then verified by the function *IsStable(I)*, which checks whether I is a minimal model of the program $\mathcal{P}^I$ (obtained by applying the GL-transformation w.r.t. I). This part of the computation is also referred to as *model check* or *stability checker*. The underlying task is co-NP-complete (cf. [11]).

```
Function DetCons(I: Interpretation): Interpretation;
(* Extend $I$ with literals that can be deterministically inferred and return the
   resulting interpretation or the set of all literals $\mathcal{L}$ upon inconsistency.
   Possible implementations are described in [5, 12, 15]. *)

Procedure Select(I: Interpretation, var L: Literal);
(* Select an undefined ground literal $L$ according to a heuristic (see Section 4). *)

Function ModelGenerator(I: Interpretation): Boolean;
begin
        I := DetCons(I);
        if I = $\mathcal{L}$ then (* inconsistency *)
            return false;
        endif
        if no atom is undefined in I then
            return IsStable(I); (* stability check *)
        endif
        Select(I,L);
        if ModelGenerator($I \cup \{L\}$) then
            return true;
        else
            return ModelGenerator($I \cup \{\texttt{not}.L\}$);
        endif
end;
```

Fig. 2 Computation of stable models in DLV

Note again that the interpretations handled by the model generator are partial interpretations where any literal is either one of true, false, or undefined w.r.t. an interpretation I.

The *ModelGenerator* function is first called with parameter I set to the empty interpretation (all atoms are undefined at this stage). If the program $\mathcal{P}$ has a stable model, then the function returns true setting I to the computed stable model; otherwise it returns false. We observe that the model generator is similar to the Davis–Putnam procedure, variants of which are frequently employed by SAT solvers. It first calls a function *DetCons(I)* which returns the extension of I with those 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 DLP for making further inferences, e.g., it exploits the knowledge that every stable model is a minimal model. [5, 12, 15]

If *DetCons* does not detect any inconsistency, a literal L is selected according to a heuristic criterion (by a call to the *Select* procedure) and *ModelGenerator* is called on both $I \cup \{L\}$ and $I \cup \{\texttt{not}.L\}$. The literal L corresponds to a *branching variable* in SAT solvers. And indeed, like for SAT solvers, the selection of a "good" literal L is crucial for the performance of a DLP system.

Remark 1 On hard DLP programs (non-HCF programs), a very large part of the computation time may be consumed by function *isStable(I)* which performs a co-NP-complete task if the program is non-HCF.

In the following, we introduce the framework for evaluating heuristic criteria as adopted in the DLV system. In Section 4 we then describe a number of heuristic criteria for the selection of such branching literals.

3.2 Evaluation of heuristic functions

The heuristic of DLV is a "dynamic heuristic" (the DLP equivalent of UP heuristics for SAT), that is, the heuristic value of a literal L depends on the result of taking L true as well as false and computing its consequences, respectively. In order to reduce the number of look-aheads, the DLV system does not evaluate the heuristic value of *all* undefined literals; rather, it considers only a subset of the undefined literals called *possibly-true* literals. The correctness of this strategy, adopted since the first release of DLV, has been shown in [27].

Definition 1 (PT literal) Let I be a partial interpretation for the (ground) program $\mathcal{P}$.

A *positive Possibly-True (PT) literal* of $\mathcal{P}$ w.r.t. I is an undefined positive literal l such that there exists a rule $r \in \mathcal{P}$ for which all of the following conditions hold:

1. l is in the head of r: $l \in H(r)$;
2. The head of r is not true w.r.t. I: $H(r) \cap I = \emptyset$;
3. The body of r is true w.r.t. I: $B(r) \subseteq I$.

A *negative PT literal* of $\mathcal{P}$ w.r.t. I is an undefined negative literal $\texttt{not}\ l$ such that there exists a rule $r \in \mathcal{P}$ for which all of the following conditions hold:

1. $\texttt{not}\ l$ is in the body of r: $\texttt{not}\ l \in B(r)$;
2. The head of r is not true w.r.t. I: $H(r) \cap I = \emptyset$;

3. The positive body of r is true w.r.t. I: $B^+(r) \subseteq I$;
4. No negative body literal is false w.r.t. I: $I \cap \texttt{not}.B^-(r) = \emptyset$.

The set of all PT literals of $\mathcal{P}$ w.r.t. I is denoted by $PT_{\mathcal{P}}(I)$.

Example 6 Consider the program $\mathcal{P}_6 = \{a \vee b \text{:-}\, c, d., e \text{:-}\, d, \texttt{not}\, f.\}$ and let $I = \{c, d\}$ be an interpretation for $\mathcal{P}_6$, then $PT_{\mathcal{P}_6}(I) = \{a, b, \texttt{not}\ f\}$.

It is worthwhile noting that the PT literals do not always restrict the set of literals to be looked-ahead, since *all* undefined literals are PT in some cases. For instance, in the program encoding 3SAT (see Section 6) every undefined literal is a PT literal, as it occurs in the head of a rule having a true (empty) body. In contrast, in the program HAMPATH, at any given stage of the computation, the PTs are those literals of the form $inPath(a, b)$ or $outPath(a, b)$, where a is a node already reached from the start ($reached(a)$ is true) and (a, b) is an arc of the input graph.

The general procedure evaluating the heuristically best literal is shown in Fig. 3. Roughly, each PT literal A is considered (**foreach** statement). Look-aheads for $I \cup \{A\}$ and for $I \cup \{\texttt{not}.A\}$ are performed (calls to *DetCons*), and results are stored in $\mathrm{I}_{\mathrm{A}}^+$ and $\mathrm{I}_{\mathrm{A}}^-$, respectively. If either the assumption of A or the assumption of $\texttt{not}.A$ leads to an inconsistency, then the complementary literal is deterministically added to the interpretation (also calling *DetCons*). Otherwise, A is compared with the previously best literal L by exploiting a heuristic criterion h_C.

Once all PTs have been considered, the best literal according to the heuristic is returned in the parameter L and then assumed in the *ModelGenerator* Function.

In the next section we describe a number of heuristic criteria that can be exploited for selecting the "best" branching literal.

```
Procedure Select(var $I$: Interpretation, var $L$: Literal);
var $\mathrm{I}_{\mathrm{A}}^+, \mathrm{I}_{\mathrm{A}}^-$: Interpretation;
begin
    $L := NULL$;
    foreach $A \in PT_{\mathcal{P}}(I)$ do
        $\mathrm{I}_{\mathrm{A}}^+ :=$ DetCons($I \cup \{A\}$);    (* look-ahead for $A$ *)
        if $\mathrm{I}_{\mathrm{A}}^+ = \mathcal{L}$ then
            $I := DetCons(I \cup \{\texttt{not}.A\})$;
        else
            $\mathrm{I}_{\mathrm{A}}^- :=$ DetCons($I \cup \{\texttt{not}.A\}$); (* look-ahead for $\texttt{not}.A$ *)
            if $\mathrm{I}_{\mathrm{A}}^- = \mathcal{L}$ then
                $I := I \cup \{A\}$;
            endif
        endif
        if $\mathrm{I}_{\mathrm{A}}^+ \neq \mathcal{L}$ and $\mathrm{I}_{\mathrm{A}}^- \neq \mathcal{L}$ then    (* no inconsistency has arisen *)
            if $L = NULL$ then
                $L := A$; (* first literal, no comparison *)
            elseif $L <_{h_C} A$ then  (* compare $A$ against L w.r.t. the heuristic *)
                $L := A$;
            endif
        endif
    endfor
end;
```

Fig. 3 Framework for the selection of the branching literal in DLV

4 Heuristics

Throughout this section, we assume that a ground program $\mathcal{P}$ and a partial interpretation I for $\mathcal{P}$ have been fixed. Here, we describe the heuristic criteria that will be compared in Section 6. We assume that the framework described in Section 3 is employed, and in particular that "dynamic heuristics" are evaluated using *Select* as in Fig. 3. Therefore, in order to define a heuristic for this framework, it is sufficient to define an appropriate comparison operator $<_{h_C}$, which may depend on $\mathrm{I}_{\mathrm{A}}^{+}$ and/or $\mathrm{I}_{\mathrm{A}}^{-}$, or rather on some values that have been collected during the computation of $\mathrm{I}_{\mathrm{A}}^{+}$ and/or $\mathrm{I}_{\mathrm{A}}^{-}$.

Given a literal L, from now we denote with $ext(I, L)$ the interpretation resulting from the application of a deterministic consequence operator on $I \cup \{L\}$. In DLV, this amounts to a call to *DetCons* (see Section 3), i.e. $ext(I, L) = DetCons(I \cup L)$. We usually assume that $ext(I, L)$ is consistent, otherwise the framework for heuristic evaluation of DLV (see Fig. 3) deterministically assumes $\mathtt{not}.L$ and the heuristic is not evaluated on L at all.

Heuristic $\mathbf{h_1}$ This is an extension of the branching rule adopted in the system SATZ [28]—an efficient SAT solver—to the framework of DLP.

The *length* of a rule r (w.r.t. an interpretation I), is the number of undefined literals occurring in r. Let $Unsat_k(L)$ denote the number of unsatisfied rules[5] of length k w.r.t. $ext(I, L)$, which have a greater length w.r.t. I. In other words, $Unsat_k(L)$ is the number of unsatisfied rules whose length shrinks to k if L is assumed and propagated in the interpretation I. The weight $w_1(L)$ is:

$$w_1(L) = \Sigma_{k>1}\, Unsat_k(L) * 5^{-k}$$

Thus, the weight function w_1 prefers literals introducing a higher number of short unsatisfied rules. Intuitively, the introduction of a high number of short unsatisfied rules is preferred because it creates more and stronger constraints on the interpretation so that a contradiction can be found earlier [28]. The factor 5 has been taken over directly from [28], where it is justified as being "empirically optimal," which we could confirm in informal tests also in our setting. We combine the weight of L with the weight of its complement $\mathtt{not}.L$ to favour L such that $w_1(L)$ and $w_1(\mathtt{not}.L)$ are roughly equal, in order to avoid that a possible failure leads to a very bad state. To this end, as in SATZ, we define the combined weight $comb\text{-}w_1(L)$ of L as follows:

$$comb\text{-}w_1(L) = w_1(L) * w_1(\mathtt{not}.L) * 1{,}024 + w_1(L) + w_1(\mathtt{not}.L).$$

The idea is that for values $w_1(L)$ and $w_1(\mathtt{not}.L)$ which are closer together, the product is greater than for values which are further apart but have the same sum. The weighting factor 1,024 is a "magic number," which has been experimentally established in [28] as being optimal. We have informally performed tests with other factors and also obtained that 1,024 performs well.

[5]Recall that a rule r is satisfied w.r.t. an interpretation J if the body of r is false w.r.t. J or the head of r is true w.r.t. J.

Given two literals A and B, heuristic h_1 prefers B over A ($A <_{h1} B$) if:

1. $comb\text{-}w_1(A) < comb\text{-}w_1(B)$ when $A \neq \texttt{not}.B$;
2. $w_1(A) < w_1(B)$ when $A = \texttt{not}.B$.[6]

Example 7 Consider the following program

$$\mathcal{P}_7 = \{r_1 : a_1 \vee na_1., \ r_2 : a_2 \vee na_2., \ r_3 : a_3 \vee na_3., \ r_4 : a_4 \vee na_4., \\ c_1 : :\!-a_1, a_2, a_3., \ c_2 : :\!-na_1, a_2, a_3., \ c_3 : :\!-a_1, a_4, a_3.\},$$

and let the current interpretation I be {}. We have eight PTs, one for each atom in the program. The heuristic will select either a_1 or na_1 which are $<_{h_1}$-maxima; indeed, both a_1 and na_1 are the only PTs having combined weight greater than zero. This happens because when we assume a_1 two unsatisfied constraints (c_1 and c_3) shrink from length 3 to 2 [i.e. $Unsat_2(a_1) = 2$]; moreover, if we assume $\texttt{not}\ a_1$ constraint c_2 shrinks from length 3 to 2 [i.e. $Unsat_2(\texttt{not}\ a_1) = 1$].[7] On the other hand, we have that for $i > 1$ $Unsat_2(\texttt{not}\ a_i) = Unsat_2(\texttt{not}\ na_i) = 0$ hold (no unsatisfied rule changes its length).

Heuristic $\mathbf{h_2}$ The second heuristic we consider is inspired by the branching rule of Smodels [43]. Let $|J|$ denote the number of literals in a (three-valued) interpretation J. Then, define

$$w_2(L) = |ext(I, L)|.$$

Since w_2 maximizes the size of the resulting interpretation, it minimizes the literals which are left undefined. Intuitively, this minimizes the size of the remaining search space [43] [which is 2^u, where u is the number of undefined atoms w.r.t. $ext(I, L)$]. Similar to Smodels, the heuristic h_2 cautiously maximizes the minimum of $w_2(L)$ and $w_2(\texttt{not}.L)$. More precisely, the preference relationship $<_{h2}$ of h_2 is defined as follows. Given two literals A and B:

1. $A <_{h2} B$ if $min(w_2(A), w_2(\texttt{not}.A)) < min(w_2(B), w_2(\texttt{not}.B))$;
2. Otherwise, $A <_{h2} B$ if $min(w_2(A), w_2(\texttt{not}.A)) = min(w_2(B), w_2(\texttt{not}.B))$, and $max(w_2(A), w_2(\texttt{not}.A)) < max(w_2(B), w_2(\texttt{not}.B))$

Example 8 Consider $\mathcal{P}_8 = \{a \vee b \vee c., \ d \vee e\!:\!-\texttt{not}\ a., \ f\!:\!-\texttt{not}\ a.\}$, and let the current interpretation I be {}; We have three PT literals a, b and c (d, e and f are not PT because the positive body of the second rule contains an undefined literal). We observe that if either b or c are assumed, only $\texttt{not}\ a$ and f are derived, while nothing is derived from the second rule. However, if a is assumed we obtain a total interpretation. In particular, we obtain $ext(I, a) = \{a, \texttt{not}\ b, \texttt{not}\ c, \texttt{not}\ d, \texttt{not}\ e, \texttt{not}\ f\}$, $ext(I, b) = \{\texttt{not}\ a, b, \texttt{not}c, f\}$, $ext(I, c) = \{\texttt{not}\ a, \texttt{not}\ b, c, f\}$. Concerning the complementary literals, we observe that assuming $\texttt{not}\ a$ can derive f, while nothing can be derived by assuming $\texttt{not}\ b$ or $\texttt{not}\ c$. In particular, we have $ext(I, \texttt{not}\ a) = \{\texttt{not}\ a, f\}$, $ext(I, \texttt{not}\ b) = \{\texttt{not}\ b\}$, and $ext(I, \texttt{not}\ c) = \{\texttt{not}\ c\}$. Looking at the

[6]Note that $comb\text{-}w_1(A) = comb\text{-}w_1(B)$ if $A = \texttt{not}.B$.

[7]It is easy to see that the same happens for $\texttt{not}\ na_1$ and na_1, respectively.

counters we have that $w_2(a) = 6$ and $w_2(b) = w_2(c) = 4$, while $w_2(\mathtt{not}\ a) = 2$ and $w_2(\mathtt{not}\ b) = w_2(\mathtt{not}\ c) = 1$. Therefore a is the best choice w.r.t. heuristic h_2. Note that a is really the best choice, since by assuming it we immediately reach a stable model, while another choice is needed if we assume either b or c.

Remark 2 It is worthwhile noting that the heuristic of Smodels, while following the above intuition, is somewhat more sophisticated than h_2. Unfortunately, it is defined for non-disjunctive programs, and centered around properties of unstratified negation. We do not see any better extension of Smodels' heuristic to the framework of disjunctive programs.

Heuristic $\mathbf{h_3}$ Let us consider now the heuristic originally used in the DLV system. Even if this is more "naïve" than the previous heuristics, we will benchmark it in order to evaluate the impact of changing the branching rule on the test system.

A peculiar property of stable models is *supportedness*, cf. Section 2.3. Since a DLP system must eventually converge to a supported interpretation, it makes sense to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, we count the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule (further details on UTs can be found in [12, 15] where they are called must-be-true atoms or MBTs). For instance, the rule *:–not x.* implies that *x* must be true in every stable model of the program; but it does not give a "support" for *x*. Thus, *DetCons* derives *x* in order to satisfy the rule and adds it to the set *UnsupportedTrue*. It will be removed from this set once a supporting rule for *x* is found (e.g., $x \vee b$:–c. would be a supporting rule for x in the interpretation $I = \{x, \mathtt{not}\ b, c\}$). Given a literal L, let $UT(L)$ be the number of UT atoms in $ext(I, L)$. Moreover, let $UT_2(L)$ and $UT_3(L)$ be the number of UT atoms occurring, respectively, in the heads of exactly 2 and 3 unsatisfied rules w.r.t. $ext(I, L)$. Intuitively, these are the most constrained UTs. Note that a UT atom in the head of only one unsatisfied rule will cause *DetCons* to make appropriate derivations in order to turn that rule into a supporting rule, therefore such atoms will not occur anymore during the evaluation of the heuristic.

The heuristic h_3 of DLV considers $UT(L)$, $UT_2(L)$ and $UT_3(L)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(L)$ of rules which are satisfied w.r.t. $ext(I, L)$. More precisely, given two literals A and B:

1. $A <_{h3} B$ if $UT(A) > UT(B)$;
2. Otherwise, $A <_{h3} B$ if $UT(A) = UT(B)$ and $UT_2(A) > UT_2(B)$;
3. Otherwise, $A <_{h3} B$ if $UT(A) = UT(B)$, $UT_2(A) = UT_2(B)$ and $UT_3(A) > UT_3(B)$;
4. Otherwise, $A <_{h3} B$ if $UT(A) = UT(B)$, $UT_2(A) = UT_2(B)$, $UT_3(A) = UT_3(B)$ and $Sat(A) < Sat(B)$.

Note that, unlike the previous heuristics, h_3 is not balanced [i.e., $ext(I, \mathtt{not}.L)$ is not considered in the heuristic]. Therefore $L <_{h3} \mathtt{not}.L$ or $\mathtt{not}.L <_{h3} L$ may hold if both L and $\mathtt{not}.L$ are PT, which does not occur for h_1 and h_2.

Example 9 Consider $\mathcal{P}_9 = \{a \vee b \vee c.,\ d \vee e \vee f.,\ {:-}\texttt{not}\ w.,\ w{:-}a.,\ w{:-}d.,\ a \vee z{:-}w.,\ b \vee z{:-}w.,\ {:-}d, z.,\ {:-}a, z.\}$, and let the current interpretation I be $\{w\}$. Here, w is UT. a and d are the $<_{h_3}$-maxima, as only assuming their truth can eliminate the UT w. Indeed, anything apart from a or d would be a poor choice.

Heuristic $\mathbf{h_4}$ The unsupported true atoms are, in a sense, the hardest constraints occurring in a DLP program. Indeed, as pointed out above, an unsupported true atom x is intuitively like a unary constraint ${:-}\texttt{not}\ x.$, which must be satisfied. By minimizing the UT atoms and maximizing the satisfied rules, the heuristic h_3 tries to drive the DLV computation toward a *supported model* (i.e., all rules are satisfied and no UT exists). Intuitively, supported models have good chances to be stable models (while unsupported models are guaranteed to be not stable models), and, for simple classes of programs (e.g., tight stratified disjunctive programs) the supported models are precisely the stable models. If the program is not tight and stratified, then supported models are not guaranteed to be stable models; but stability checking can be done efficiently if the program is HCF.

For hard programs (i.e., non-HCF programs), supported models are often not stable models. Stability checking is computationally expensive (co-NP-complete), and may consume a large portion of the resources needed for computing a stable model.

The heuristic h_4, described next, tries to elaborate on h_3 and drives the computation toward supported models having higher chances to be stable models, with the goal of reducing the overall number of the expensive stability checks. Models having a higher degree of supportedness are preferred, where the degree of supportedness is the average number of supporting rules for a true atom (note that this number is greater or equal to 1 for supported models). Intuitively, if all true atoms have many supporting rules in a model M, then the elimination of an atom from the model would violate many rules. In this case, it becomes less likely that a subset of M exists, which is a model of $\mathcal{P}^M$ and would thus disprove that M is a stable model. The idea for h_4 is thus to prefer choices which lead to a greater degree of supportedness.

We next formalize this intuition. Given a literal L, let $True(L)$ be the number of true non-HCF atoms in $ext(I, L)$, and let $SuppRules(L)$ be the number of all supporting rules for non-HCF atoms w.r.t. $ext(I, L)$. Let DS be the degree of supportedness of the interpretation intended as the ratio between the number of supporting rules and the number of true atoms [i.e. $DS(L) = SuppRules(L)/True(L)$[8]]. Note that heuristic h_4 is "balanced" (i.e. it takes into account both the look-ahead for the branching literal and its complement) and is defined as a refinement of the heuristic h_3.

To this end, given a literal L, let $UT'(L) = UT(L) + UT(\texttt{not}.L)$, $UT_2'(L) = UT_2(L) + UT_2(\texttt{not}.L)$, $UT_3'(L) = UT_3(L) + UT_3(\texttt{not}.L)$, $Sat'(L) = Sat(L) + Sat(\texttt{not}.L)$, and $DS'(L) = DS(L) + DS(\texttt{not}.L)$.

[8]In the implementation, the denominator is increased by 1, in order to avoid possible divisions by zero.

The preference relationship $<_{h4}$ of h_4 is defined as follows. Given two literals A and B:

1. $A <_{h4} B$ if $UT'(A) > UT'(B)$;
2. Otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$ and $UT'_2(A) > UT'_2(B)$;
3. Otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$ and $UT'_3(A) > UT'_3(B)$;
4. Otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$, $UT'_3(A) = UT'_3(B)$ and $Sat'(A) < Sat'(B)$.
5. Otherwise, $A <_{h4} B$ if $UT'(A) = UT'(B)$, $UT'_2(A) = UT'_2(B)$, $UT'_3(A) = UT'_3(B)$, $Sat'(A) = Sat'(B)$ and $DS'(A) < DS'(B)$.

Example 10 Reconsider Example 9 where we assumed that $I = \{w\}$. We get $ext(I, a) = \{w, a, b, \texttt{not}\ z, \texttt{not}\ c\}$, $ext(I, d) = \{w, d, a, b, \texttt{not}\ z, \texttt{not}\ c, \texttt{not}\ e, \texttt{not}\ f\}$. $DS(a) = 3/3$, since $w{:-}a$; $a \vee z{:-}w$ and $b \vee z{:-}w$ are supporting rules for the three true non-HCF atoms w, a, b. On the other hand, $DS(d) = 4/3$, since $w{:-}d$ is an additional supporting rule for the same three true non-HCF atoms w, a, b. Since $DS(\texttt{not}\ a) = DS(\texttt{not}\ d) = 2/2$, we obtain $a <_{h_4} d$. Indeed, d is arguably a better choice than a. It immediately leads to a stable model whereas a would require at least another choice, and choosing e or f would cause a failing stability check.

5 Optimizing the computation of heuristics

In the previous section we described a number of "dynamic" heuristics for deciding which branching literal to assume. To select the "best" next branching literal the system looks ahead by tentatively assuming each undefined (PT) literal L and its complement. Then the heuristic value of L [which is a measure of the "quality" of the resulting interpretations $ext(I, L)$ and $ext(I, \texttt{not}.L)$] is exploited to select the next branching literal.

As will be shown in Section 6.3.1, some of the heuristics are very effective, as they drastically reduce the number of choice points arising in the computation of stable models. However, the computation of these heuristics often is quite expensive since the number of (PT) literals to be considered is very large in some cases, and the cost of a look-ahead is linear or quadratic in the size of the Herbrand Base in the worst case. The computation of the heuristics thus often consumes most of the total time taken by a DLP system.

In this section, we present two techniques that reduce the amount of time needed to evaluate the heuristics by reducing the number of look-aheads that need to be performed.

5.1 Look-ahead equivalences

Dynamic heuristics depend only on the interpretations resulting from $ext(I, L)$ and $ext(I, \texttt{not}.L)$. We would therefore like to identify cases in which two literals L and L' are *look-ahead equivalent*, i.e., $ext(I, L) = ext(I, L')$, since only one of the two look-ahead computations has to be done in order to obtain the heuristic value for both L and L' then. This notion of equivalence is formalized next.

Definition 2 Let p and q be two undefined literals w.r.t. an interpretation I and *ext* a function mapping interpretations to interpretations (plus $\mathcal{L}$). p and q are *look-ahead equivalent* with respect to *ext* if $ext(I, p) = ext(I, q)$.

To single out a sufficient and efficiently checkable condition which guarantees such an equivalence, we first define the notion of a *potentially supporting rule*:

Definition 3 Given a program $\mathcal{P}$, an atom a, and a (three-valued) interpretation I, a rule $r \in \mathcal{P}$ is a *potentially supporting rule* for a w.r.t. I, if the following conditions are satisfied: (1) a occurs in the head of r, (2) no atom in $H(r) \setminus \{a\}$ is true w.r.t. I, and (3) no literal in the body of r is false w.r.t. I. Let $psupp_{\mathcal{P}}(a, I)$ denote the number of potentially supporting rules for a.

The optimization which we describe next actually does not depend on a particular version or implementation of *ext* which is used. In order to formulate a minimal assumption on what should be computed by *ext* such that the optimization is applicable, we introduce the disjunctive extension of the classic inflationary $T_{\mathcal{P}}$ operator.

Definition 4 Given a ground program $\mathcal{P}$ and an interpretation I, we define

$$\begin{aligned}\mathcal{T}_{\mathcal{P}}(I) = I \cup \{a \mid r \in \mathcal{P}, a \in H(r),\ &\text{the body of } r \text{ is true w.r.t. } I,\\ &\text{the head of } r \text{ apart from } a \text{ is false w.r.t. } I\}\end{aligned}$$

Furthermore we employ the disjunctive inflationary version of Fitting's operator.

Definition 5 Given a ground program $\mathcal{P}$ and an interpretation I, let

$$\phi_{\mathcal{P}}(I) = I \cup \{\texttt{not}\ a \mid psupp_{\mathcal{P}}(a, I) = 0\}$$

Moreover, we assume that *ext* computes a fixpoint and introduce a property called *saturating*.

Definition 6 Given a ground program $\mathcal{P}$, an interpretation I, and a function *ext* mapping interpretations to interpretations (in the context of assuming a literal L), *ext* is called *saturating* if $a \in ext(I, L)$ implies $ext(I, L) = ext(I \cup \{a\}, L)$.

We can now formulate some look-ahead equivalence results.

Proposition 3 *Let ext be a saturating operator such that for any interpretation I it holds that $\mathcal{T}_{\mathcal{P}}(I \cup \{L\}) \subseteq ext(I, L)$ and $\phi_{\mathcal{P}}(I \cup \{L\}) \subseteq ext(I, L)$. If two undefined positive literals a and b occur in the head of a rule r in a program $\mathcal{P}$, and a and b are* *the only undefined literals w.r.t. an interpretation I in r (where we assume that there is no multiple occurrence of atoms in rules), then it holds that:*

1. *If $psupp_{\mathcal{P}}(b, I) = 1$, a and* $\texttt{not}\ b$ *are look-ahead equivalent w.r.t. ext.*
2. *If $psupp_{\mathcal{P}}(a, I) = 1$,* $\texttt{not}\ a$ *and b are look-ahead equivalent w.r.t. ext.*

Proof Suppose that a and b are the only undefined literals w.r.t. an interpretation I in r and that $psupp_{\mathcal{P}}(b, I) = 1$. Therefore $psupp_{\mathcal{P}}(b, I \cup \{a\}) = 0$, hence $\texttt{not}\ b \in \phi_{\mathcal{P}}(I \cup \{a\})$ and $\texttt{not}\ b \in ext(I, a)$. Since *ext* is saturating, we have $ext(I, a) = ext(I \cup \{\texttt{not}\ b\}, a)$. On the other hand, the body of r is true w.r.t. $I \cup \{\texttt{not}\ b\}$ and the head apart from a is false w.r.t. $I \cup \{\texttt{not}\ b\}$, so $a \in \mathcal{T}_{\mathcal{P}}(I \cup \{\texttt{not}\ b\})$ and $a \in ext(I, \texttt{not}\ b)$. Because *ext* is saturating we obtain $ext(I, \texttt{not}\ b) = ext(I \cup \{a\}, \texttt{not}\ b)$. In total, we have $ext(I, a) = ext(I \cup \{\texttt{not}\ b\}, a) = ext(I \cup \{a\}, \texttt{not}\ b) = ext(I, \texttt{not}\ b)$. It follows that a and $\texttt{not}\ b$ are look-ahead equivalent w.r.t. *ext*, thus proving item 1.

Symmetrically, suppose that a and b are the only undefined literals w.r.t. an interpretation I in r and that $psupp_{\mathcal{P}}(a, I) = 1$. It is easy to see that this implies $psupp_{\mathcal{P}}(b, I \cup \{b\}) = 0$, hence $\texttt{not}\ a \in \phi_{\mathcal{P}}(I \cup \{b\})$ and $\texttt{not}\ a \in ext(I, b)$. Since *ext* is saturating, we have $ext(I, b) = ext(I \cup \{\texttt{not}\ a\}, b)$. On the other hand, the body of r is true w.r.t. $I \cup \{\texttt{not}\ a\}$ and the head apart from b is false w.r.t. $I \cup \{\texttt{not}\ a\}$, so $b \in \mathcal{T}_{\mathcal{P}}(I \cup \{\texttt{not}\ a\})$ and $b \in ext(I, \texttt{not}\ a)$. Because *ext* is saturating we obtain $ext(I, \texttt{not}\ a) = ext(I \cup \{b\}, \texttt{not}\ a)$. In total, we have $ext(I, b) = ext(I \cup \{\texttt{not}\ a\}, b) = ext(I \cup \{b\}, \texttt{not}\ a) = ext(I, \texttt{not}\ a)$, proving item 2. □

Corollary 1 *Proposition 3 holds if ext is a version of DetCons as defined in* [5], [12], *or* [15].

Example 11 Consider the program $\{a \vee b.\}$ and $I = \emptyset$. Both a and b are PT literals, so look-ahead for a, $\texttt{not}\ a$, b, and $\texttt{not}\ b$ is performed, i.e. we compute $ext(I, a) = \{a, \texttt{not}\ b\}$, $ext(I, \texttt{not}\ a) = \{\texttt{not}\ a, b\}$, $ext(I, b) = \{\texttt{not}\ a, b\}$, and $ext(I, \texttt{not}\ b) = \{a, \texttt{not}\ b\}$. In this example we can save the look-aheads for $\texttt{not}\ b$ and b because of Proposition 3, and thus save half of the look-aheads.

In the implementation of DLV we recognize the applicability of Proposition 3 very efficiently and avoid superfluous look-aheads. Experimental results reported in Section 6 will show that we can avoid up to 50% of look-aheads in some practical cases (e.g. on 3SAT) by exploiting this simple condition.

5.2 Two-layered heuristics

In [28] a different idea on reducing look-aheads has been presented: An easy-to-compute heuristic is defined as a first layer, and look-ahead is only computed on those possible choices which look promising in that regard. This gives a two-layered heuristics. While the optimization exploiting look-ahead equivalence was exact in the sense that the choices and computation trees remain precisely the same, this optimization is somewhat fuzzy as the simple heuristic in the first layer may cut away the best literal according to the more expensive second layer heuristic. The properties of these two optimizations are therefore quite different.

The simple heuristic criterion defined in [28] involves the number of *binary clauses* a literal occurs in. The rationale is that this is the number of immediate propagations that can be performed during look-ahead. This idea can be directly transferred to the DLP framework:

Definition 7 A *binary clause* is a rule which contains exactly two undefined literals w.r.t. an interpretation I. The *number of binary occurrences* of an undefined literal l is the number of binary clauses l occurs in.

Note that this notion directly corresponds to the number of immediate propagations which can be performed by assuming l and `not` l, so it matches the intuition of [28]. To reduce the number of literals to be looked-ahead, we adopt the following criterion:

First-Layer Heuristic S_{bin} Let $PT_{\mathcal{P}}(I)$ be the set of PT literals of a ground program $\mathcal{P}$ w.r.t. an interpretation I, and let $S_{bin} \subseteq PT_{\mathcal{P}}(I)$ be the set of PT literals having more than the average number of binary occurrences w.r.t. all literals in $PT_{\mathcal{P}}(I)$. Then, consider only the literals in S_{bin} for the selection of the branching literals (i.e., perform look-ahead only on these literals).

Note that our first-layer heuristic is inspired by the same intuition as the first-layer heuristic in [28], even though it is not precisely the same due to the different setting.

6 Experiments and benchmarks

In this section, we first describe in detail the benchmark problems and instances considered. Then we comparatively analyze the heuristics from Section 4, and finally evaluate the impact of the optimization techniques described in Section 5.

6.1 Benchmark problems

We evaluate the considered heuristics and the proposed optimizations on the following problems:

- Hamiltonian Path (HAMPATH)
- Satisfiability (3SAT)
- Strategic Companies (STRATCOMP)
- Quantified Boolean Formulae (2QBF)

The first two benchmarks (HAMPATH and 3SAT) are well-known NP-complete problems, which have been frequently used to assess the efficiency of DLP systems (see, e.g., [25, 37]). Since our goal is also the evaluation of heuristics for *disjunctive* logic programming systems, we consider two additional benchmarks (STRATCOMP and 2QBF) that strictly require the use of disjunction, as their complexity is located at the second level of the polynomial hierarchy.

HAMPATH is a classical NP-complete problem from the area of graph theory:

> Given an undirected graph $G = (V, E)$, where V is the set of vertices of G and E is the set of edges, and a node $a \in V$ of this graph, does there exist a path of G starting at a and passing through each node in V exactly once?

Suppose that the graph G is specified by using two predicates $node(X)$ and $arc(X, Y)$[9], and the starting node is specified by the unary predicate $start(X)$ which contains only a single tuple. Then, the following program solves the problem HAMPATH:

```
% Guess whether to take a path or not.
$inPath(X, Y) \vee outPath(X, Y) :- reached(X), arc(X, Y).$

% Reach nodes starting from the given one.
$reached(X) :- start(X).$
$reached(X) :- inPath(Y, X).$

% Each node has to be reached.
$:- node(X), \texttt{not}\ reached(X).$

% At most one incoming/outgoing arc!
$:- inPath(X, Y), inPath(X, Y1), Y <> Y1.$
$:- inPath(X, Y), inPath(X1, Y), X <> X1.$
```

Basically, the disjunctive rule guesses a subset S of the arcs to be in the path, while the rest of the program checks whether S constitutes a Hamiltonian Path. By means of the first rule, the auxiliary predicate *reached* (modeling nodes that can be reached) influences the guess of *inPath*, which is done in a gradual way: Initially, the second rule is applied, marking the starting node as reached. Then a guess among arcs leaving the starting node is made, followed by repeated guesses of arcs leaving from nodes marked as reached by the third rule, until all reachable nodes have been handled. The first constraint enforces that all nodes in the graph are reached from the starting node in the subgraph induced by S. The other two constraints ensure that the set of arcs S selected by *inPath* meets the following requirements, which any Hamiltonian Path must satisfy: (1) there must not be two arcs starting at the same node, and (2) there must not be two arcs ending in the same node.

3SAT is a special case of SAT, one of the best researched problems in AI and frequently used for solving many other problems by translating them to SAT, solving the SAT problem, and transforming the solution back to the original domain.

> Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee d_{i,2} \vee d_{i,3})$ where the $d_{i,\cdot}$ are literals over propositional variables $x_1, \ldots, x_m$.
> Φ is satisfiable, iff there exists a consistent conjunction I of literals such that $I \models \Phi$.

3SAT is a classical NP-complete problem [38] and can be easily represented in DLP as follows:

For every propositional variable x_i ($1 \leq i \leq m$), we add the following rule which ensures that we either assume that variable x_i or its complement nx_i is true: $x_i \vee nx_i$.

[9]Predicate *arc* is symmetric, since undirected arcs are bidirectional.

And for every clause $d_1 \vee d_2 \vee d_3$ in Φ we add the constraint $:- \bar{d}_1, \bar{d}_2, \bar{d}_3$. where $\bar{d}_k$ $(1 \leq k \leq 3)$ is nx_j if $d_k = x_j$, and $\bar{d}_k = x_j$ if $d_k = \neg x_j$.

STRATCOMP is a Σ_2^P-complete problem, which has first been described in [4]:

A holding owns companies $C(1), \ldots, C(c)$, each of which produces some goods. Some of these companies may jointly control another one.
Now, some companies should be sold, under the constraint that all goods can still be produced, and that no company is sold which would still be controlled by the holding afterwards. A company is strategic, if it belongs to a *strategic set*, which is a minimal set of companies satisfying these constraints.

Computing strategic companies is Σ_2^P-hard in general [4]. Reformulated as a decision problem ("Given a particular company c in the input, is c strategic?"), it is Σ_2^P-complete.

As in [4] we assume that each product is produced by at most two companies and each company is jointly controlled by at most three companies. In this settings, an instance of STRATCOMP can be modeled by asserting the following facts:

- A fact $produced\_by(P, C1, C2)$ whenever product P is produced by companies $C1$ and $C2$;
- A fact $controlled\_by(C, C1, C2, C3)$ whenever company C is jointly controlled by $C1$, $C2$ and $C3$

The following program solves this hard problem elegantly by only the following two rules [25].

$$strategic(C1) \vee strategic(C2) :- produced\_by(P, C1, C2).$$

$$\begin{aligned} strategic(C) :- {} & controlled\_by(C, C1, C2, C3), \\ & strategic(C1), strategic(C2), strategic(C3). \end{aligned}$$

Here $strategic(C)$ means that company C is a strategic company.

Basically, the program exploits the minimization which is inherent to the semantics of stable models for the check whether a candidate set C' of companies that produces all goods and obeys company control is also minimal with respect to this property. In particular, the disjunctive rule intuitively guesses one of the companies c_1 and c_2 that produce some item p; while the second rule checks that no company is sold that would be controlled by other companies in the strategic set, by simply requesting that this company must be strategic as well. The stable models of this program correspond one-to-one to the strategic sets, indeed a company c is strategic iff $strategic(c)$ is in some stable model of the program [25]. The minimality of the strategic sets is automatically ensured by the minimality of stable models.

Checking whether any given company c_i is strategic is done by brave reasoning: "Is there *any* stable model containing c_i?"

2QBF is the canonical problem for the second level of the polynomial hierarchy [38].

Let Φ be a quantified Boolean formula (QBF) of the form $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$.
Check whether Φ evaluates to true.

We used a transformation from 2QBF to disjunctive logic programming which is a slightly altered form of a reduction used in [9]. 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 is made as follows:

- For every variable $v_i \in X \cup Y$ add a rule $v_i \vee nv_i.$;
- For every variable $y \in Y$ add a couple of rules of the form $y{:}{-}w.\ ny{:}{-}w.$;
- For every conjunction $l_1 \wedge l_2 \wedge l_3$ in ϕ add a rule $w{:}{-}\bar{l}_1, \bar{l}_2, \bar{l}_3.$ where $\bar{l} = l$ if l is a variable and $\bar{l} = nv$ if l is $\neg v$, and
- Add the constraint :−`not` w;

The 2QBF formula Φ is valid iff $\mathcal{P}_\Phi$ is satisfiable [9].

6.2 Benchmark data

The instances for HAMPATH were generated by a tool by Patrik Simons which has been used to compare Smodels against SAT solvers (cf. [42]). For each problem size n we generated 20 instances, always assuming node 1 as the starting node.

Concerning the instances for 3SAT, we have randomly generated 3CNF formulas over n variables using a tool by Selman and Kautz [41]. Again, for each size we generated 20 such instances, where we kept the ratio between the number of clauses and the number of variables at 4.3, which is near the cross-over point for random 3SAT [6].

For STRATCOMP, we randomly generated 20 instances for each problem size n, with n companies and n products. Each company O is controlled by one to five companies (two groups of companies, where each of these groups controls the same company O, must have at least one member in common), where the actual number of companies is uniform randomly chosen. On average this results in 1.5 *controlled_by* relations per company.

In case of 2QBF, finally, we generated 100 random 2QBF instances for each problem size, according to the method presented in [22] where transition phase results for QBFs are reported. In all generated 2QBF instances, the number of $\forall$-variables in any formula is the same as the number of $\exists$-variables (that is, $|X| = |Y|$) and each disjunct contains at least two universal variables. Moreover, the number of clauses is $((|X| + |Y|)/2)^{0.5}$.

It is worthwhile noting that these numbers of clauses have been chosen according to the experimental verification reported in [22] (see [25], for a discussion on this issue).

6.3 Experimental results

The experiments have been performed on a machine equipped with two Intel Xeon HT CPUs clocked at 3.60GHz with 2 MB of Level 2 Cache and 3GB of RAM, running Debian GNU Linux (kernel 2.4.27-2-686-smp). The binaries were generated with GCC 3.4.

We have allowed at most 7,200 s (2 h) of execution time for each instance, and we set a memory usage limit to 1 GB of memory for each process (by exploiting the

Fig. 4 Comparison of the heuristics: average execution time and number of look-aheads ▶

ulimit command). The experimentation was stopped (for each system) at the size at which some instance exceeded this time limit[10].

The executables and input files used for the benchmarks are available on the web at http://www.mat.unical.it/ricca/downloads/amai07.tar.gz.

The results of our experiments are displayed in the graphs of Figs. 4 and 5. For each problem domain we report two graphs: In both graphs the horizontal axis reports a parameter representing the size of the instance, while on the vertical axis we report the running time (expressed in seconds) and the number of look-aheads, respectively, averaged over the instances of the same size we have run (see previous section). The details of the experimental results are reported in the Appendix.

6.3.1 Comparison of the heuristics

The results obtained for comparing the relative efficiencies of the heuristics h_1 to h_4 are reported in Fig. 4.

In the graphs displaying the benchmark results, the line of a heuristic stops before the largest instance size whenever some problem instance was not solved within the allowed time limit.

Table 1 displays, for each heuristic, the maximum instance-size where the heuristic could solve *all* problem instances in the maximum allowed time; and, the "winner" heuristic (i.e. the heuristic solving the largest instances, or, in case two heuristics solved the same instances, the one taking the lowest average time on the largest instances) is outlined in bold face.

As expected, heuristic h_3, the earliest look-ahead heuristic of the DLV system, which is "unbalanced" (i.e., it does not combine the heuristic values of complementary atoms), is the worst in most cases. Indeed, apart from the case of 2QBF, where it performs slightly better than both h_1 and h_2, heuristic h_3 is outperformed by all the competitors; moreover, it could solve STRATCOMP instances only up to 2,400 companies (taking 3,553.4 s where h_4 required at most 10.9 s).

Heuristic h_1, the extension of SATZ heuristic to DLP, behaves quite well on average. As expected (this heuristic was defined for SAT), h_1 is the fastest on 3SAT. It behaves well also on STRATCOMP, where h_1 was the second fastest and, as h_2 and h_4, it could solve all benchmark instances we have run. It shows suboptimal behaviour on HAMPATH and, especially, on 2QBF, where considering unsupported true atoms and (for 2QBF) the degree of support seems to be crucial for the efficiency.

Heuristic h_4 behaves (somehow expectedly) much better than its "predecessor" h_3. It is "balanced" (the heuristic values of the positive and of the negative literal are combined by sum) and, it is tailored for hard (Σ_2^P-complete) problems (in addition to unsupported true, it considers also the degree of support).

[10]Actually, the memory usage limit had never been exceeded.

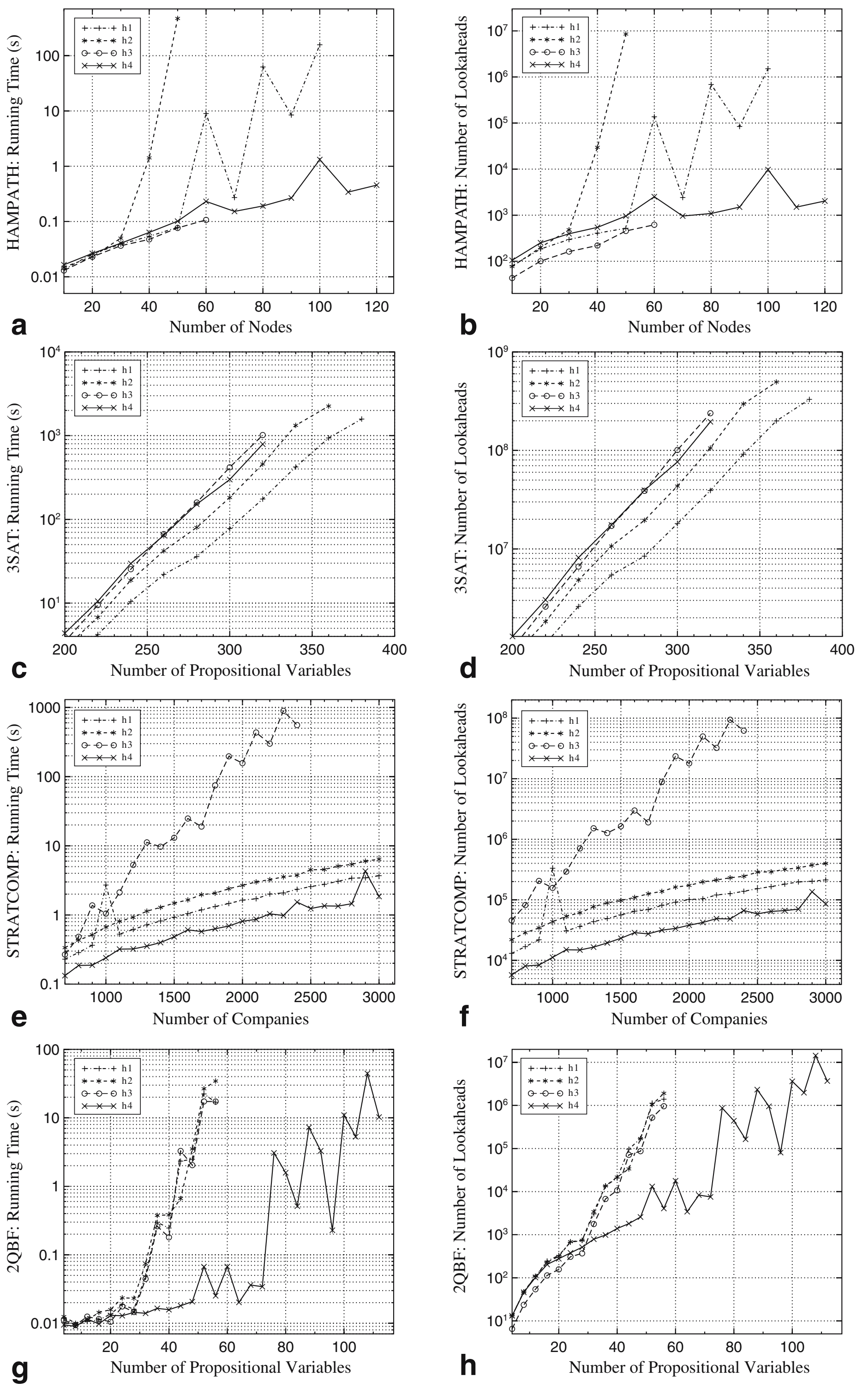

h1
h2
h3
h4
HAMPATH: Running Time (s)
Number of Nodes
a
HAMPATH: Number of Lookaheads
Number of Nodes
b
3SAT: Running Time (s)
Number of Propositional Variables
c
3SAT: Number of Lookaheads
Number of Propositional Variables
d
STRATCOMP: Running Time (s)
Number of Companies
e
STRATCOMP: Number of Lookaheads
Number of Companies
f
2QBF: Running Time (s)
Number of Propositional Variables
g
2QBF: Number of Lookaheads
Number of Propositional Variables
h

Fig. 5 Impact of the optimizations: average execution time and number of look-aheads ▶

Thanks to these refinements, heuristic h_4 dramatically improves the performance of h_3. Indeed, heuristic h_4 is the best one in nearly all problems we ran (only on 3SAT it stopped earlier than other heuristics). Importantly, h_4 is the best heuristic on the hardest problems (STRATCOMP and 2QBF), and it beats all other heuristics by a relevant factor on both HAMPATH and 2QBF. In particular, in the former problem the second best (h_1) stopped at 110, while h_4 could solve instances up to 130 nodes within the time limit; moreover, to solve an instance of size 110, h_4 averagely takes 1.3s, while h_1 takes 157.4 s. The performance gap is even more significant in the case of 2QBF, where the heuristics different from h_4 can solve instances up to size 56 (within the time limit), while h_4 can solve all qbf problems having up to 112 propositional variables (twice the size of all the others). Here, the second best (h_3) took 425.12 s to solve instances of size 56, an enormity compared with the 0.47 s required by h_4.

The behaviour of heuristic h_2, based on the minimization of the undefined atoms, is rather controversial. It behaves well on 3SAT, but it is rather bad on HAMPATH, where it stops at 50 nodes already, and is beaten even by the "naive" heuristic h_3. This confirms that further studies are needed to find a proper extension of the heuristic of Smodels to the setting of disjunctive programs.

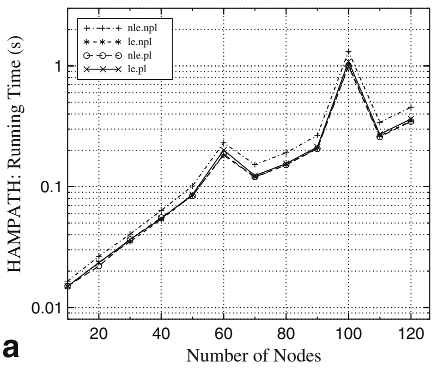
Concluding, we observe that both heuristic h_1 and heuristic h_4 significantly improve the efficiency of the native heuristic h_3 of the DLV system. While h_1 is the fastest in 3SAT, heuristic h_4 is the best performing in all the remaining problems, being orders of magnitude faster than all the competitors in both HAMPATH and 2QBF.

6.3.2 Impact of the optimization techniques

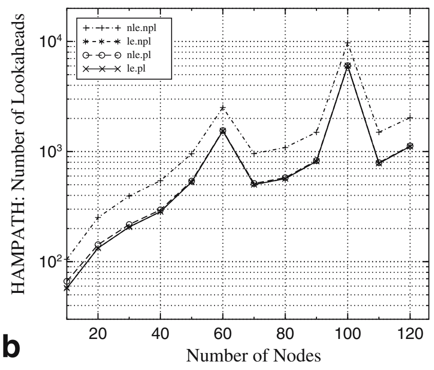
As previously pointed out, heuristic functions can help to significantly reduce the number of choices made by the DLP system, and consequently they can increase the performance of DLP systems. However, most of the execution time is spent by systems for "looking-ahead". The graphs of Fig. 4 confirm this observation. Indeed, the lines reporting the execution times have nearly the same "shape" of the (corresponding) ones reporting the number of look-aheads performed, demonstrating that execution times directly correlate with the number of look-ahead performed.

Table 1 Maximum instance-size where a heuristic could solve all problem instances in the maximum allowed time

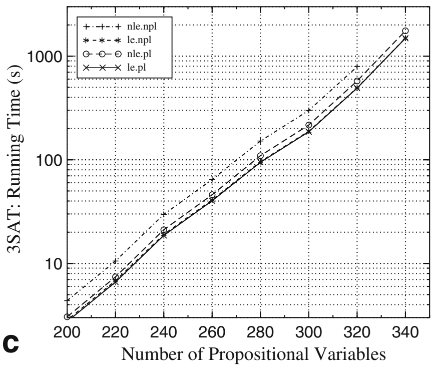
Prob./Heur.	h_1	h_2	h_3	h_4
HAMPATH	110	50	60	**130**
3SAT	**400**	380	340	340
STRATCOMP	>3,000	>3,000	2,400	**>3,000**
2QBF	56	56	56	**112**


nle.npl
le.npl
nle.pl
le.pl
HAMPATH: Running Time (s)
Number of Nodes

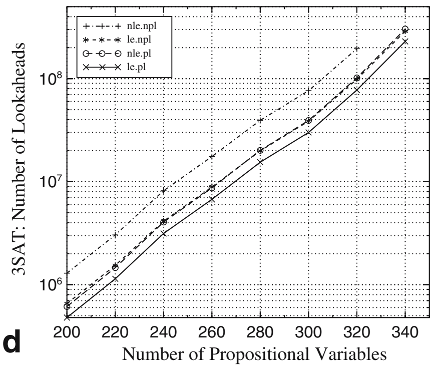
a


nle.npl
le.npl
nle.pl
le.pl
HAMPATH: Number of Lookaheads
Number of Nodes

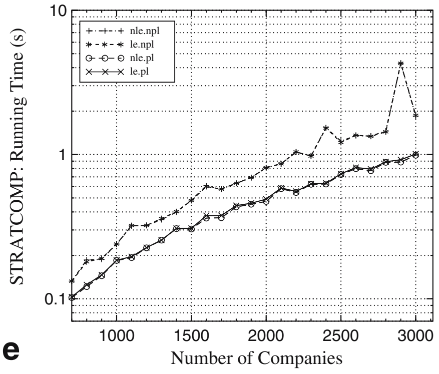
b


nle.npl
le.npl
nle.pl
le.pl
3SAT: Running Time (s)
Number of Propositional Variables

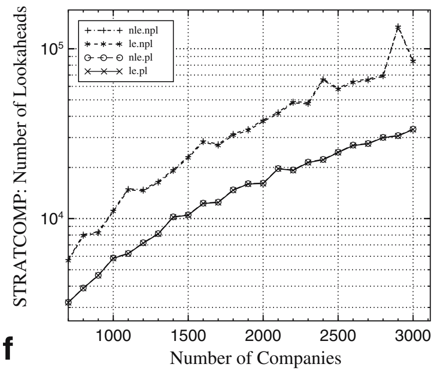
c


nle.npl
le.npl
nle.pl
le.pl
3SAT: Number of Lookaheads
Number of Propositional Variables

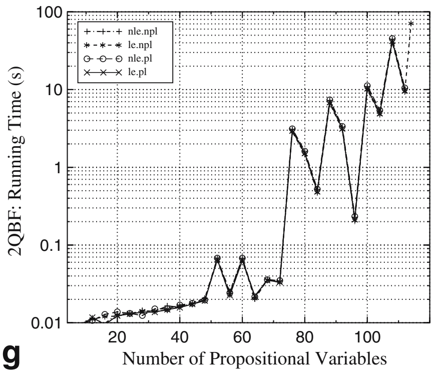
d


nle.npl
le.npl
nle.pl
le.pl
STRATCOMP: Running Time (s)
Number of Companies

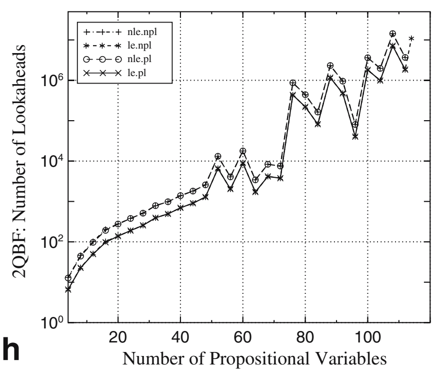
e


nle.npl
le.npl
nle.pl
le.pl
STRATCOMP: Number of Lookaheads
Number of Companies

f


nle.npl
le.npl
nle.pl
le.pl
2QBF: Running Time (s)
Number of Propositional Variables

g


nle.npl
le.npl
nle.pl
le.pl
2QBF: Number of Lookaheads
Number of Propositional Variables

h

We have experimentally evaluated the impact of the two optimization techniques described in Section 5, which are aimed at reducing the number of look-ahead, in order to improve the system performance. In particular, we have selected the best heuristic resulting from the experiments of the previous section (namely, heuristic h_4), and we have evaluated the impact of the two optimization techniques on our collection benchmark problems. The results of the experiments are displayed in Fig. 5. In the graphs, the curves labeled by "nle.npl", "le.npl", "nle.pl", and "le.pl" denote, respectively, the initial (unoptimized) version (i.e., precisely heuristic h_4 as benchmarked in the previous subsection), the look-ahead equivalence optimization, the two-layered optimization (pl stands for pre-look-ahead), and the combination of both look-ahead equivalence and two-layered optimization (all versions adopt the same heuristic h_4). Again, the line of a version stops before the largest instance size whenever some problem instance was not solved within the allotted time limit.

Observe first that both optimizations always bring some gain over the original version, as the "no optimization" curve is always on top of the other three curves in all graphs. In addition, there are cases where the "optimized" versions of the system could increase the maximum size of instances solved within the time limit, (e.g. for 3SAT le.pl could solve instances having up to 340 variables, while the unoptimized nle.npl could reach only size 320).

The two optimizations have different impact, depending on the problem domain: For 2QBF, the equivalence optimization performs better than the two-layered approach, while for Strategic Companies and 3SAT the opposite holds. For Hamiltonian Path both optimizations behave roughly equal.

In most cases, the combination of the two optimizations combines the benefits in the sense that performance is as good as for the best of the two strategies. Indeed, apart from 2QBF, where le-npl is clearly the best option (it has been the only one able to solve all the instances we provided), the curve combining the two strategies (le.pl) nearly coincides with the curve of the best of le.npl and nle.pl (e.g. for Strategic Companies le.pl and nle.pl actually overlap). On Hamiltonian Path le.npl, nle.pl, and le.pl all give the same speed-up, while in the case of 3SAT there are even better results for the "combined" le.pl than for any of the two methods alone: the combination even sums up the individual gains of the two optimizations here!

Note that when the two-layered option is exploited (nle.pl and le.pl), the runtime and the number of look-aheads need not correlate, as fewer look-aheads are performed but the quality of the PTs may be worse, which may lead to larger computation trees (this seems to be crucial in the case of 2QBF). On the other hand, for look-ahead equivalence the choices remain the same, but only the number of look-aheads can be reduced, so avoided look-aheads directly reduce the runtime in this case. This can be easily noted in the graphs reporting the number of look-aheads performed where the line of nle.npl is never on top of le.npl.

It is worthwhile noting that the gap between the original version and the best option, which is really impressive in some cases (e.g. for an instance of 2000 companies of STRATCOMP the average number of look-aheads steps from 38045 to 16090), grows exponentially. Indeed, the gain appears to be nearly constant in the graphs, in which we have employed a logarithmic scale.

In conclusion, both optimizations turned out to be useful, and the combination of the two seems to be the best choice since the introduced benefits combine well in most cases.

7 Related work and conclusion

In this paper, we have studied the use of look-ahead heuristics to enhance the efficiency of a DLP system. We have introduced a number of heuristic criteria for DLP and we have also described two optimization techniques for reducing the cost of their computation.

Specifically, we have designed, implemented and evaluated several *look-ahead* heuristics for DLP systems. Among these, some are extensions to DLP of successful heuristics for SAT solvers or non-disjunctive systems, while others are custom-designed for DLP, being geared towards hard problems on the second level of the polynomial hierarchy.

Since the computational core of a DLP system is somewhat similar to a SAT-solver, the literature on the design and comparison of heuristics for SAT [7, 19, 23, 24, 28–30, 36] is related to our work. In particular, the works dealing with "unit propagation" [UP] heuristics [19, 24, 28–30] are directly related to our techniques, since look-ahead heuristics are the DLP equivalent of UP heuristics for SAT.

While a lot of work has been done for SAT in this area, only little is available for logic programming systems. A couple of works have dealt with heuristics for non-disjunctive logic programs [20, 43], and we are not aware of any previous comparison between look-ahead heuristics for DLP, apart from preliminary versions of the present work.

Since the computation of look-ahead heuristics is often very expensive, we have also introduced two methods which aim at reducing the amount of time needed by the system to evaluate the heuristics. The first one is based on a condition which is sufficient to guarantee that two literals have precisely the same heuristic value, thus being able to avoid look-ahead for one of them. The second one is a two-layered heuristic (a computationally cheap heuristic criterion reduces the set of literals to be considered for look-ahead) similar to the one employed in the SAT solver SATZ [28].

Some techniques for reducing the number of look-aheads have been employed also in SAT solvers and in other LP systems. In particular, Smodels makes a drastic pruning of the look-aheads by eliminating each literal which has been derived during a previous look-ahead at the same branch point: For each literal $B \in ext(I, A)$, the look-ahead for B is not performed in that case because B is guaranteed to be worse than A w.r.t. the heuristic function of Smodels. This technique eliminates a high number of look-aheads. It is not generally applicable, though, since it relies on a monotonicity property of the heuristic in that it requires that $ext(I, B) \subset ext(I, A)$ implies that B is worse than A w.r.t. the heuristic. Our technique avoids a smaller number of look-aheads, but it is applicable to every criterion determining the heuristic value from the result of the look-ahead (i.e., its applicability requires that the heuristic value of A depends only on $ext(I, A)$), and is therefore very general. In fact, our technique can also be applied in Smodels, while the optimization employed by Smodels cannot be used in DLV since the heuristic employed in DLV is not monotonic in the sense described above.

We have implemented these heuristics and optimization techniques in the state-of-the-art system DLV and we have compared their respective efficiency on a number of benchmark problems taken from various domains. The experiments show interesting results and evidence a couple of promising heuristic criteria for DLP. In particular, we verified that the newly proposed heuristic h_4 significantly improves the performance of DLV by consistently reducing the average execution time, and, in some cases, by enlarging the maximum solvable size of considered problems for a fixed time limit.

In addition, both proposed optimization techniques proved to be useful and demonstrated to be somehow "orthogonal" to each other, since their integration performs at least as well as the best individual technique, resulting in a relevant improvement of the performance of the DLV system. In particular, the time spent for looking-ahead often decreases by an exponential factor w.r.t. the size of the instance to be solved.

We believe that this paper is not at all a conclusive work on heuristics for DLP. Rather, it is a first step in this field that will hopefully stimulate further work on the design and evaluation of heuristics for DLP, which are strongly needed to build efficient DLP solvers. Ongoing work concerns the design and experimentation of "look-back" heuristics for DLP, preliminary results in this direction are reported in [13].

Acknowledgements This work was 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."

Appendix: Experimental results

In Tables 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, and 17, we report all the results of the experiments.

Table 2 Running time (HAMPATH)

HAMPATH	Running time			
Nodes	h_1	h_2	h_3	h_4
10	0.01	0.02	0.01	0.02
20	0.03	0.02	0.02	0.03
30	0.04	0.05	0.04	0.04
40	0.05	1,416.00	0.05	0.06
50	0.08	4,715,785.00	0.08	0.10
60	89,895.00	–	0.11	0.23
70	0.27	–	–	0.15
80	624,185.00	–	–	0.19
90	8,444.00	–	–	0.27
100	157,409.00	–	–	1,318.00
110	–	–	–	0.34
120	–	–	–	0.46

Table 3 Number of look-aheads (HAMPATH)

HAMPATH	Number of look-aheads			
Nodes	h_1	h_2	h_3	h_4
10	79.35	76.70	43.25	105.15
20	184.65	200.15	100.40	251.90
30	296.00	482.40	162.20	395.00
40	406.75	29,725.15	219.70	543.85
50	519.85	8,564,268.85	454.40	956.65
60	136,628.85	–	618.70	2,516.85
70	2,419.50	–	–	959.25
80	687,010.15	–	–	1,089.35
90	84,250.50	–	–	1,498.05
100	1,509,390.80	–	–	9,730.75
110	–	–	–	1,498.75
120	–	–	–	2,032.85

Table 4 Running time (3SAT)

3SAT	Running time			
Variables	h_1	h_2	h_3	h_4
200	19,195.00	28,745.00	34,875.00	43,725.00
220	4,224.00	67,725.00	9,481.00	105,415.00
240	104,375.00	18,747.00	25,425.00	29,709.00
260	22,033.00	42,075.00	66,682.00	64,428.00
280	359,915.00	80,237.00	1,589,445.00	151,774.00
300	778,575.00	1,823,165.00	418,283.00	299,256.00
320	1,749,885.00	4,585,625.00	10,172,035.00	7,914,295.00
340	4,228,155.00	1,327.11	–	–
360	9,386,135.00	2,252,265.00	–	–
380	1,579,625.00	–	–	–

Table 5 Number of look-aheads (3SAT)

3SAT	Number of look-aheads			
Variables	h_1	h_2	h_3	h_4
200	514,541.70	795,779.00	977,001.65	1,292,602.15
220	1,098,561.25	1,825,065.95	2,594,416.40	3,038,620.50
240	2,606,440.10	4,823,409.50	6,615,051.75	8,165,741.60
260	5,432,957.90	10,693,855.90	17,141,645.70	17,529,559.85
280	8,443,585.00	19,506,935.00	38,855,545.50	39,383,455.25
300	18,081,627.30	43,433,842.00	101,069,224.40	76,431,371.60
320	39,188,497.45	105,771,076.50	238,706,623.20	195,322,397.70
340	91,660,077.00	295,788,065.10	–	–
360	199,284,838.60	495,248,261.90	–	–
380	329,516,744.50	–	–	–

Table 6 Running time (STRATCOMP)

STRATCOMP	Running time			
Companies	h_1	h_2	h_3	h_4
700	0.23	0.34	0.27	0.13
800	0.29	0.43	0.48	0.19
900	0.37	0.52	1,373.00	0.19
1,000	2,705.00	0.67	1,047.00	0.24
1,100	0.53	0.81	2,115.00	0.32
1,200	0.62	0.92	5,287.00	0.32
1,300	0.72	1,134.00	111,995.00	0.35
1,400	0.82	12,875.00	96,865.00	0.40
1,500	0.92	1,481.00	13,054.00	0.48
1,600	10,465.00	1.65	247,975.00	0.61
1,700	11,845.00	1,964.00	189,625.00	0.58
1,800	13,225.00	20,635.00	74,577.00	0.63
1,900	14,645.00	2,395.00	1,963,045.00	0.69
2,000	16,365.00	2,676.00	1,567,705.00	0.81
2,100	17,135.00	29,985.00	434.19	0.86
2,200	20,075.00	32,415.00	299.56	10,375.00
2,300	2,083.00	35,495.00	8,931,435.00	0.98
2,400	23,495.00	3,738.00	5,560,345.00	15,335.00
2,500	2,569.00	44,955.00	–	12,385.00
2,600	27,775.00	45,385.00	–	13,525.00
2,700	30,625.00	5,073.00	–	13425.00
2,800	34,025.00	54,015.00	–	14,545.00
2,900	3,442.00	5,992.00	–	4,293.00
3,000	36,915.00	64,085.00	–	1.85

Table 7 Number of look-aheads (STRATCOMP)

STRATCOMP	Number of look-aheads			
Companies	h_1	h_2	h_3	h_4
700	13,031.40	21,929.05	45,650.95	5,735.80
800	16,852.75	28,628.55	81,811.40	8,060.35
900	21,875.50	34,335.70	204,873.75	8,360.30
1,000	332,296.20	43,511.20	157,318.10	11,204.85
1,100	30,823.95	53,457.30	292,091.45	14,958.75
1,200	36,160.30	60,980.35	706,348.40	14,829.30
1,300	43,622.20	76,758.95	1,517,438.90	16,539.05
1,400	49,097.35	88,730.75	1,267,056.70	19,404.15
1,500	56,237.30	97,674.55	1,638,255.70	23,159.20
1,600	64,874.45	109,316.95	2,983,529.75	28,657.30
1,700	68,839.35	125,757.05	1,899,242.85	27,420.65
1,800	81,121.40	137,760.10	8,877,178.85	31,548.00
1,900	91,075.50	161,964.35	23,483,449.20	33,674.40

Table 7 (continued)

STRATCOMP	Number of look-aheads			
Companies	h_1	h_2	h_3	h_4
2,000	101,156.95	173,019.50	17,946,809.60	38,045.15
2,100	103,774.60	197,531.30	49,823,881.10	42,271.25
2,200	120,292.15	212,608.70	32,262,209.70	48,830.60
2,300	126,411.15	231,114.50	94,124,164.55	48,334.50
2,400	138,313.85	244,560.10	62,033,350.30	66,393.05
2,500	152,924.85	286,094.30	–	58,562.85
2,600	165,031.60	289,985.60	–	64,365.45
2,700	181,622.15	318,022.55	–	66,289.10
2,800	197,537.45	336,483.00	–	70,120.45
2,900	202,460.30	373,462.60	–	135,650.40
3,000	212,902.45	397,690.35	–	85,673.65

Table 8 Running time (2QBF)

2QBF	Running time			
Variables	h_1	h_2	h_3	h_4
4	0.01	0.01	0.01	0.01
8	0.01	0.01	0.01	0.01
12	0.01	0.01	0.01	0.01
16	0.01	0.01	0.01	0.01
20	0.01	0.02	0.01	0.01
24	0.02	0.02	0.02	0.01
28	0.02	0.02	0.01	0.01
32	0.05	0.07	0.04	0.01
36	0.30	0.38	0.26	0.02
40	0.25	0.38	0.18	0.02
44	23,334.00	0.66	32,824.00	0.02
48	24,811.00	35,131.00	20,384.00	0.02
52	223,282.00	265,495.00	173,356.00	0.07
56	16,884.00	344,734.00	172,966.00	0.03
60	–	–	–	0.07
64	–	–	–	0.02
68	–	–	–	0.04
72	–	–	–	0.03
76	–	–	–	30,688.00
80	–	–	–	15,778.00
84	–	–	–	0.51
88	–	–	–	72,453.00
92	–	–	–	32,962.00
96	–	–	–	0.23
100	–	–	–	109,595.00
104	–	–	–	52,949.00
108	–	–	–	444,365.00
112	–	–	–	102,539.00

Table 9 Number of look-aheads (2QBF)

2QBF	Number of look-aheads			
Variables	h_1	h_2	h_3	h_4
4	13.24	13.04	6.52	13.10
8	47.08	47.16	23.58	46.08
12	106.56	108.68	54.22	101.72
16	244.88	229.28	113.52	203.88
20	330.56	314.40	156.84	277.04
24	669.72	687.28	307.72	379.12
28	733.60	742.28	368.74	512.40
32	3,040.84	3,417.36	1,768.20	787.00
36	13,534.00	13,545.84	6,754.06	982.80
40	20,653.24	21,751.40	10,703.06	1,392.08
44	96,369.88	33,646.40	71,258.36	1,806.88
48	171,224.16	170,043.60	86,794.20	2,573.92
52	1,129,738.32	1,043,890.68	522,107.62	13,135.96
56	1,392,740.76	1,910,637.04	958,009.30	4,064.52
60	–	–	–	17,808.16
64	–	–	–	3,425.52
68	–	–	–	8,326.52
72	–	–	–	7,530.52
76	–	–	–	867,439.12
80	–	–	–	440,212.76
84	–	–	–	164,039.12
88	–	–	–	2,323,932.52
92	–	–	–	952,475.08
96	–	–	–	80,552.52
100	–	–	–	3,618,697.28
104	–	–	–	1,975,514.12
108	–	–	–	14,342,517.28
112	–	–	–	3,684,184.80

Table 10 Running time (HAMPATH)

HAMPATH	Running time			
Nodes	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
10	0.02	0.02	0.02	0.02
20	0.03	0.02	0.02	0.02
30	0.04	0.04	0.04	0.04
40	0.06	0.05	0.06	0.05
50	0.10	0.09	0.08	0.09
60	0.23	0.18	0.19	0.20
70	0.15	0.12	0.12	0.12
80	0.19	0.15	0.15	0.16
90	0.27	0.21	0.21	0.21
100	1,318.00	10,125.00	0.99	10,885.00
110	0.34	0.26	0.26	0.27
120	0.46	0.35	0.35	0.36

Table 11 Number of look-aheads (HAMPATH)

HAMPATH	Number of look-aheads			
Nodes	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
10	105.15	57.55	66.35	57.55
20	251.90	132.90	142.70	132.90
30	395.00	206.90	217.50	206.70
40	543.85	284.75	295.80	284.60
50	956.65	529.15	539.90	529.00
60	2,516.85	1,541.35	1,552.75	1,541.05
70	959.25	501.75	513.75	501.55
80	1,089.35	567.95	579.85	567.75
90	1,498.05	816.95	828.80	816.80
100	9,730.75	6,028.55	6,040.35	6,028.35
110	1,498.75	782.45	793.25	782.25
120	2,032.85	1,115.15	1,126.20	1,114.80

Table 12 Running time (3SAT)

3SAT	Running time			
Variables	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
200	43,725.00	28,355.00	30,525.00	27,445.00
220	105,415.00	6,813.00	7,415.00	6.61
240	29,709.00	19,054.00	21,127.00	18,617.00
260	64,428.00	41,016.00	459,785.00	40,218.00
280	151,774.00	960,185.00	109,531.00	945,885.00
300	299,256.00	1,887,345.00	216,654.00	186.38
320	7,914,295.00	4,948,535.00	575,797.00	491,587.00
340		1,493,447.00	17,560,215.00	1,489,379.00

Table 13 Number of look-aheads (3SAT)

3SAT	Number of look-aheads			
Variables	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
200	1,292,602.15	655,452.35	612,828.40	479,154.00
220	3,038,620.50	1,539,643.70	1,461,513.85	1,139,557.65
240	8,165,741.60	4,134,930.20	4,050,509.00	3,139,523.40
260	17,529,559.85	8,869,320.35	8,706,728.95	6,733,015.45
280	39,383,455.25	19,920,812.65	20,129,210.15	15,450,580.35
300	76,431,371.60	38,635,094.50	39,267,676.20	30,041,871.00
320	195,322,397.70	98,691,034.80	101,875,220.60	77,573,921.70
340		288,756,198.70	303,486,353.80	229,903,935.70

Table 14 Running time (STRATCOMP)

STRATCOMP	Running time			
Companies	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
700	0.13	0.13	0.10	0.10
800	0.19	0.18	0.12	0.13
900	0.19	0.19	0.14	0.15
1,000	0.24	0.24	0.18	0.19
1,100	0.32	0.32	0.19	0.20
1,200	0.32	0.32	0.23	0.23
1,300	0.35	0.36	0.26	0.26
1,400	0.40	0.40	0.31	0.31
1,500	0.48	0.48	0.30	0.31
1,600	0.61	0.60	0.36	0.38
1,700	0.58	0.58	0.36	0.38
1,800	0.63	0.63	0.43	0.44
1,900	0.69	0.69	0.45	0.46
2,000	0.81	0.81	0.47	0.49
2,100	0.86	0.86	0.58	0.59
2,200	10,375.00	1,043.00	0.55	0.56
2,300	0.98	0.98	0.62	0.63
2,400	15,335.00	1,529.00	0.62	0.63
2,500	12,385.00	1,212.00	0.73	0.74
2,600	13,525.00	13,595.00	0.80	0.81
2,700	13,425.00	1,337.00	0.77	0.79
2,800	14,545.00	1,437.00	0.88	0.89
2,900	4,293.00	4,292.00	0.89	0.92
3,000	1.85	1,869.00	0.99	10,085.00

Table 15 Number of look-aheads (STRATCOMP)

STRATCOMP	Number of look-aheads			
Companies	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
700	5,735.80	5,650.15	3,217.90	3,205.70
800	8,060.35	7,921.45	3,903.05	3,892.15
900	8,360.30	8,222.45	4,637.85	4,625.95
1,000	11,204.85	11,018.65	5,865.60	5,837.15
1,100	14,958.75	14,750.40	6,233.80	6,219.45
1,200	14,829.30	14,589.55	7,209.30	7,173.50
1,300	16,539.05	16,278.80	8,153.40	8,123.75
1,400	19,404.15	19,073.40	10,237.85	10,200.40
1,500	23,159.20	22,781.00	10,529.40	10,478.70
1,600	28,657.30	28,195.65	12,328.80	12,280.75
1,700	27,420.65	26,906.55	12,500.10	12,435.45
1,800	31,548.00	30,961.90	14,756.15	14,704.00
1,900	33,674.40	32,990.90	16,052.90	15,981.85

Table 15 (continued)

STRATCOMP	Number of look-aheads			
Companies	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
2,000	38,045.15	37,309.70	16,165.05	16,090.95
2,100	42,271.25	41,445.65	19,703.20	19,653.40
2,200	48,830.60	48,021.35	19,322.55	19,227.75
2,300	48,334.50	47,312.10	21,500.60	21,381.20
2,400	66,393.05	65,436.50	22,300.80	22,184.85
2,500	58,562.85	57,553.60	24,590.80	24,494.80
2,600	64,365.45	63,173.40	27,096.30	26,920.30
2,700	66,289.10	65,087.45	27,662.65	27,571.85
2,800	70,120.45	68,819.15	30,066.35	29,971.10
2,900	135,650.40	133,928.70	30,800.35	30,649.55
3,000	85,673.65	84,038.60	33,627.30	33,469.70

Table 16 Running time (2QBF)

2QBF	Running time			
Variables	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
4	0.01	0.01	0.01	0.01
8	0.01	0.01	0.01	0.01
12	0.01	0.01	0.01	0.01
16	0.01	0.01	0.01	0.01
20	0.01	0.01	0.01	0.01
24	0.01	0.01	0.01	0.01
28	0.01	0.01	0.01	0.01
32	0.01	0.01	0.02	0.01
36	0.02	0.01	0.02	0.01
40	0.02	0.02	0.02	0.02
44	0.02	0.02	0.02	0.02
48	0.02	0.02	0.02	0.02
52	0.07	0.06	0.07	0.07
56	0.03	0.02	0.02	0.02
60	0.07	0.06	0.07	0.06
64	0.02	0.02	0.02	0.02
68	0.04	0.04	0.04	0.04
72	0.03	0.03	0.03	0.03
76	30,688.00	28,618.00	31,274.00	29,549.00
80	15,778.00	14,719.00	16,102.00	15,212.00
84	0.51	0.47	0.52	0.49
88	72,453.00	66,394.00	73,765.00	68,642.00
92	32,962.00	30,524.00	33,556.00	31,431.00
96	0.23	0.21	0.23	0.21
100	109,595.00	9,967.00	112,119.00	103,267.00
104	52,949.00	47,661.00	54,132.00	49,446.00
108	444,365.00	408,852.00	453,752.00	42,249.00
112	102,539.00	93,693.00	104,998.00	97,255.00
114	–	709,231.00	–	–

Table 17 Number of look-aheads (2QBF)

2QBF	Number of look-aheads			
Variables	*nle.npl*	*le.npl*	*nle.pl*	*le.pl*
4	13.10	6.58	12.70	6.58
8	46.08	23.04	44.26	22.72
12	101.72	50.86	97.10	49.76
16	203.88	101.94	192.40	97.26
20	277.04	138.52	276.28	138.28
24	379.12	189.56	379.12	189.56
28	512.40	256.20	512.40	256.20
32	787.00	393.50	787.00	393.50
36	982.80	491.40	982.80	491.40
40	1,392.08	696.04	1,392.08	696.04
44	1,806.88	903.44	1,806.88	903.44
48	2,573.92	1,286.96	2,573.92	1,286.96
52	13,135.96	6,567.98	13,135.96	6,567.98
56	4,064.52	2,032.26	4,064.52	2,032.26
60	17,808.16	8,904.08	17,808.16	8,904.08
64	3,425.52	1,712.76	3,425.52	1,712.76
68	8,326.52	4,163.26	8,326.52	4,163.26
72	7,530.52	3,765.26	7,530.52	3,765.26
76	867,439.12	433,719.56	867,439.12	433,719.56
80	440,212.76	220,106.38	440,212.76	220,106.38
84	164,039.12	82,019.56	164,039.12	82,019.56
88	2,323,932.52	1,161,966.26	2,323,932.52	1,161,966.26
92	952,475.08	476,237.54	952,475.08	476,237.54
96	80,552.52	40,276.26	80,552.52	40,276.26
100	3,618,697.28	1,809,348.64	3,618,697.28	1,809,348.64
104	1,975,514.12	987,757.06	1,975,514.12	987,757.06
108	14,342,517.28	7,171,258.64	14,342,517.28	7,171,258.64
112	3,684,184.80	1,842,092.40	3,684,184.80	1,842,092.40
114	–	10,882,713.96	–	–

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Baral, C., Gelfond, M.: Logic programming and knowledge representation. J. Log. Program. **19/20**, 73–148 (1994)
3. Ben-Eliyahu, R., Dechter, R.: Propositional semantics for disjunctive logic programs. Ann. Math. Artif. Intell. **12**, 53–87 (1994)
4. Cadoli, M., Eiter, T., Gottlob, G.: Default logic as a query language. IEEE Trans. Knowl. Data Eng. **9**(3), 448–463 (1997)
5. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning operators for disjunctive logic programming systems. Fundam. Inform. **71**(2–3), 183–214 (2006)
6. Crawford, J.M., Auton, L.D.: Experimental results on the crossover point in random 3SAT. Artif. Intell. **81**(1–2), 31–57 (1996)
7. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Nebel, B. (ed.) Proc. 17th International Joint Conference on Artificial Intelligence IJCAI-01, pp. 248–253. Morgan Kaufmann (2001)

8. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative problem-solving using the DLV system. In: Minker, J. (ed.) Logic-Based Artificial Intelligence, pp. 79–103. Kluwer Academic Publishers (2000)
9. Eiter, T., Gottlob, G.: On the computational cost of disjunctive logic programming: propositional case. Ann. Math. Artif. Intell. **15**(3/4), 289–323 (1995)
10. Eiter, T., Gottlob, G.: The complexity of logic-based abduction. J. Assoc. Comput. Mach. **42**(1), 3–42 (1995)
11. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive datalog. ACM Trans. Database Syst. **22**(3), 364–418 (1997)
12. Faber, W.: Enhancing efficiency and expressiveness in answer set programming systems. Ph.D. thesis, Institut für Informationssysteme, Technische Universität Wien (2002)
13. Faber, W., Leone, N., Maratea, M., Ricca, F.: Experimenting with look-back heuristics for hard ASP programs. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) Logic Programming and Nonmonotonic Reasoning—9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 2007, Proceedings, Lecture Notes in AI (LNAI), vol. 4483, pp. 110–122. Springer Verlag (2007)
14. Faber, W., Leone, N., Mateis, C., Pfeifer, G.: Using database optimization techniques for nonmonotonic reasoning. In: INAP Organizing Committee (ed.) Proc. 7th International Workshop on Deductive Databases and Logic Programming (DDLP'99), pp. 135–139. Prolog Association of Japan (1999)
15. Faber, W., Leone, N., Pfeifer, G.: Pushing goal derivation in DLP computations. In: Gelfond, M., Leone, N., Pfeifer, G. (eds.) Proc. 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). Lecture Notes in AI (LNAI), vol. 1730, pp. 177–191. Springer Verlag, El Paso, TX, USA (1999)
16. Faber, W., Leone, N., Pfeifer, G.: Experimenting with heuristics for answer set programming. In: Proc. Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, pp. 635–640. Morgan Kaufmann Publishers, Seattle, WA, USA (2001)
17. Faber, W., Leone, N., Pfeifer, G.: Optimizing the computation of heuristics for answer set programming systems. In: Eiter, T., Faber, W., Truszczyński, M., (eds.) Logic Programming and Nonmonotonic Reasoning—6th International Conference, LPNMR'01, Vienna, Austria, September 2001, Proceedings, Lecture Notes in AI (LNAI), vol. 2173, pp. 288–301. Springer Verlag (2001)
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 International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings, Lecture Notes in Computer Science, vol. 3662, pp. 240–252. Springer Verlag (2005)
19. Freeman, J.W.: Improvements on propositional satisfiability search algorithms. Ph.D. thesis, University of Pennsylvania (1995)
20. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), pp. 386–392. Morgan Kaufmann Publishers (2007)
21. Gelfond, M., Lifschitz, V.: Classical Negation in logic programs and disjunctive databases. New Gener. Comput. **9**, 365–385 (1991)
22. Gent, I., Walsh, T.: The QSAT phase transition. In: Proc. 16th AAAI (1999)
23. Goldberg, E., Novikov, Y.: BerkMin: a fast and robust sat-solver. In: Design, Automation and Test in Europe Conference and Exposition (DATE 2002), pp. 142–149, 4–8 March 2002, Paris, France, IEEE Computer Society (2002)
24. Hooker, J.N., Vinay, V.: Branching rules for satisfiability. J. Autom. Reason. **15**, 359–383 (1995)
25. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Log. **7**(3), 499–562 (2006)
26. 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, pp. 33–42 (2001)
27. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: unfounded sets, fixpoint semantics and computation. Inf. Comput. **135**(2), 69–112 (1997)
28. Li, C., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proc. Fourteen International Joint Conference on Artificial Intelligence (IJCAI) 1997, pp. 366–371. Nagoya, Japan (1997)

29. Li, C., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Smolka, G. (ed.) Proc. Third International Conference on Principles and Practice of Constraint Programming (CP'97), Lecture Notes in Computer Science, vol. 1330, pp. 342–356. Springer (1997)
30. Li, C.M., Gérard, S.: On the limit of branching rules for hard random unsatisfiable 3-SAT. In: Proc. 14th European Conference on Artificial Intelligence ECAI-2000, pp. 98–102 (2000)
31. 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, Lecture Notes in Computer Science, vol. 3662, pp. 447–451. Springer Verlag (2005)
32. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. In: Proc. Eighteenth National Conference on Artificial Intelligence (AAAI-2002). AAAI Press, Edmonton, Alberta, Canada (2002)
33. Marek, V.W., Subrahmanian, V.: The relationship between logic program semantics and non-monotonic reasoning. In: Proc. 6th International Conference on Logic Programming—ICLP'89, pp. 600–617. MIT Press (1989)
34. Minker, J.: On indefinite data bases and the closed world assumption. In: Loveland, D.W. (ed.) Proceedings 6th Conference on Automated Deduction (CADE '82). Lecture Notes in Computer Science, vol. 138, pp. 292–308. Springer, New York (1982)
35. Minker, J.: Overview of disjunctive logic programming. Ann. Math. Artif. Intell. **12**, 1–24 (1994)
36. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proc. 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18–22, 2001, pp. 530–535. ACM (2001)
37. Niemelä, I., Simons, P.: Efficient implementation of the well-founded and stable model semantics. In: Maher, M.J. (ed.) Proc. 1996 Joint International Conference and Symposium on Logic Programming (ICLP'96). pp. 289–303. MIT Press, Bonn, Germany (1996)
38. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1994)
39. Przymusinski, T.C.: Stable semantics for disjunctive programs. New Gener. Comput. **9**, 401–424 (1991)
40. Rintanen, J.: Improvements to the evaluation of quantified boolean formulae. In: Dean, T. (ed.) Proc. Sixteenth International Joint Conference on Artificial Intelligence (IJCAI) 1999, pp. 1192–1197. Morgan Kaufmann Publishers, Stockholm, Sweden (1999)
41. Selman, B., Kautz, H.: ftp://ftp.research.att.com/dist/ai/ (1997)
42. Simons, P.: Extending and implementing the stable model semantics. Ph.D. thesis, Helsinki University of Technology, Finland (2000)
43. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. **138**, 181–234 (2002)