Optimizing the Computation of Heuristics for Answer Set Programming Systems*

Wolfgang Faber[1], Nicola Leone[2], and Gerald Pfeifer[1]

[1] Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
faber@kr.tuwien.ac.at, pfeifer@dbai.tuwien.ac.at
[2] Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
leone@unical.it

Abstract. Most SAT solvers and Answer Set Programming (ASP) systems employ a backtracking search by repeatedly assuming the truth of literals. The choice of these branching literals is crucial for the performance of these systems.
Competitive ASP systems employ advanced heuristics to select branching literals, which are usually based on "look-ahead" techniques: To evaluate the heuristic value of a literal L, truth and falsity of L are assumed in the current interpretation, consequences are derived, and the quality of the resulting interpretations is evaluated. This process can be very expensive, and often consumes most of the time taken by an ASP system.
In this paper, we present two techniques to optimize the computation of the heuristics in the ASP system DLV. The first technique singles out pairs of literals $\langle A, not\ B\rangle$ having precisely the same consequences, which allows for making only one look-ahead for each of these pairs. The second technique (inspired by SAT solvers) is a 2-layered heuristic, in which a simple heuristic criterion reduces the set of literals to be looked-ahead.
We implement both techniques in the ASP system DLV and evaluate their efficiency on a number of benchmark problems taken from various domains. The experiments confirm the usefulness of both techniques, sensibly improving the performance of DLV.

1 Introduction

DLV is a knowledge representation system based on disjunctive logic programming (DLP) [Min82,GL91] offering front-ends to several Knowledge Representation (KR) formalisms [ELM+98b,ELM+98a,EFLP99]. A strong point of DLV is its highly expressive language, which allows elegant and natural representations of very hard problems (up to Σ_2^P-hard problems). DLV supports a declarative programming style which has recently been termed *Answer Set Programming (ASP)*, hence it is referred to as an *ASP system*. The idea of ASP is to represent

* This work was supported by FWF (Austrian Science Funds) under the projects Z29-INF and P14781 and MURST under project COFIN-2000 "From Data to Information (D2I)".

a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver (like DLV) to compute them [Lif99].

An efficient support for the highly expressive language of DLV requires the use of smart algorithms and data structures as well as sophisticated optimization techniques in order to deal with such hard computational tasks.

DLV employs backtracking search by repeatedly assuming the truth of literals [FLP99], and in order to improve the efficiency of the DLV system, in a previous paper [FLP01] we have experimented with a number of heuristics for deciding which branching literal to assume. These heuristics are based on "look-ahead" techniques: to evaluate the heuristic value of a literal L w.r.t. the interpretation I at hand, truth and falsity of L are assumed in the current interpretation, and its consequences are derived by computing its deterministic extensions $I' = DetCons(I \cup \{L\})$ (the interpretation I' is guaranteed to be contained precisely in the same answer sets containing $I \cup \{L\}$) and $I'' = DetCons(I \cup \{\texttt{not } L\})$. Note that I' and I'' can be inconsistent, in which case the search space can be pruned early.

The heuristic value of L is a measure of the "quality" of the resulting interpretations I' and I''. Some of these heuristics proved to be very useful, as they drastically reduce the number of choice-points arising in an ASP computation. However, the computation of these heuristics is very expensive, since the number of literals to be "looked-ahead" is very large in some cases, and the cost of a look-ahead is linear 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 an ASP system, and may slow down the ASP system significantly.

In this paper, we try to reduce the amount of time needed to evaluate the heuristics, by reducing the number of look-aheads that need to be performed. The main contributions of the paper are the following:

A. We define a new condition which is sufficient to guarantee that, at a given stage of the computation, two literals $\langle A, not\ B\rangle$ have precisely the same set of deterministic consequences w.r.t. the interpretation I at hand, that is, $DetCons(I \cup \{A\}) = DetCons(I \cup \{not\ B\})$. Consequently, A and $not\ B$ are guaranteed to have precisely the same heuristic values, and we avoid the look-ahead for one of them.
   This technique allows us to save 50% of the look-aheads in several cases including, e.g., Hamiltonian Path and 3SAT programs.
B. We design a 2-layered heuristic. A computationally cheap heuristic criterion reduces the set of literals to be considered, and the look-ahead to select the branching literal is applied only to the literals in this set. This method significantly reduces the number of look-aheads, but, unlike the previous technique, it is not an "exact" method, that is, it might exclude literals which would otherwise have had high heuristic values. Also some literals for which the look-ahead detects inconsistency can be missed in this way, so there will be less pruning in general.
C. We implement the above techniques in the ASP system DLV and evaluate their efficiency on a number of benchmark problems taken from various do-

mains. The results of the experiments are very positive and both techniques prove to be useful. Moreover, they are orthogonal and 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 addition to the above contributions, we explain in detail the heuristic criterion adopted in DLV for the selection of the branching literal, and the way how it is computed.

It is worthwhile noting that techniques for reducing the number of look-aheads have been employed in SAT solvers and in other ASP systems. In particular, the ASP system 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 $B \in DetCons(I \cup \{A\})$, the look-ahead for B is not performed, because B is guaranteed to be worse than A w.r.t. the heuristic function of Smodels. This technique eliminates a higher number of look-aheads than our technique described in Item A; but our technique is more general and it is applicable to a wider class of heuristic functions. Indeed, the technique of Smodels relies on a monotonicity property of the heuristic: $DetCons(I \cup \{B\}) \subset DetCons(I \cup \{A\})$ implies that B is worse than A w.r.t. the heuristic function of Smodels. Our technique, instead, is applicable to every criterion determining the heuristic value from the result of the look-ahead (i.e., the heuristic value of A depends only on $DetCons(I \cup \{A\})$. 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. A 2-layered heuristic similar to the technique of Item B above has been successfully employed in the SAT solver SATZ [LA97].

2 Answer Set Programming Language

In this section, we provide a formal definition of the syntax and semantics of the ASP language supported by DLV: disjunctive datalog extended with strong negation. For further background, see [GL91,EFLP00].

ASP Programs A *(disjunctive) rule* r is a formula

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

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are classical literals (atoms possibly preceded by the classical negation symbol $\neg$) and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \cdots, b_k, \ \texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ is the *body*, $b_1, \cdots, b_k$ the *positive body*, and $\texttt{not} \ b_{k+1}, \cdots, \texttt{not} \ b_m$ the *negative body* of r. The sets of literals in head, body, positive body and negative body of r, are denoted by $H(r)$, $B(r)$, $B^+(r)$, and $B^-(r)$, respectively. Comparison operators (like $=, <, >, <>$) are built-in predicates in ASP systems, and may appear in the bodies of rules. A *disjunctive datalog program* (also called *ASP program* in this paper) $\mathcal{P}$ is a finite set of rules.

As usual, an object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables.

Answer Sets We describe the semantics of consistent answer sets, which has originally been defined in [GL91].

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 combinations of predicate symbols appearing in $\mathcal{P}$ with constants of $U_\mathcal{P}$, possibly preceded by $\neg$.

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

For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in two steps, following [Lif96]:

A set L of literals is said to be *consistent* if, for every literal $\ell \in L$, its complementary literal is not contained in L. An interpretation I is a consistent set of ground literals. An interpretation $I \subseteq B_\mathcal{P}$ is *closed under* $\mathcal{P}$ (where $\mathcal{P}$ is a positive program), if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true whenever all literals in the body are true. I is an *answer set* for $\mathcal{P}$ if it is minimal w.r.t. set inclusion and closed under $\mathcal{P}$.

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

3 Answer Set Computation

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine of the DLV system, but also other ASP systems, like Smodels employ a very similar procedure.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation of $\mathcal{P}$.[1] The hard part of the computation is then performed on this ground ASP program.

The heart of the computation is performed by the Model Generator, which is sketched in Figure 1. Roughly, the Model Generator produces some "candidate" answer sets. The stability of each of these is subsequently verified by the function *IsAnswerSet(I)*, which checks whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$, the reduct of $Ground(\mathcal{P})$ w.r.t. I.

[1] This ground instantiation is required to have precisely the same answer sets as $\mathcal{P}$, and is usually much smaller than $Ground(\mathcal{P})$ [FLMP99].

```
Forward Function DetCons(I: Interpretation): Interpretation;
(* Extends I with the literals that can be deterministically inferred and returns the
   resulting interpretation or the set of all literals 𝓛 upon inconsistency. [FLP99] *)
Forward Procedure Select(var I: Interpretation, var L: ClassicalLiteral);
(* Selects the classical literal L having the highest heuristic value (see Section 4 *)

Function ModelGenerator(var I: Interpretation): Boolean;
  (* The function returns True iff I can be extended to an answer set. *)
var   inconsistency: Boolean;
begin
      I := DetCons(I);
      if I = 𝓛 then return False;   (* inconsistency detected *)
      if no literal is undefined in I then return IsAnswerSet(I);
      Select(I,L);
      if ModelGenerator(I ∪ {L}) then return True;
      else return ModelGenerator(I ∪ {not L});
end;
```

Fig. 1. Computation of Answer Sets

The ModelGenerator function is first invoked with parameter I set to the empty interpretation.[2] If the program $\mathcal{P}$ has an answer set, the function returns True, setting I to that answer set; otherwise it returns False. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls the function DetCons, which extends I with those literals that can be deterministically inferred from I. DetCons is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, a classical literal L is selected according to a heuristic criterion by a call to the Select procedure. ModelGenerator is then recursively called on $I \cup \{L\}$; if this call does not generate an answer set (i.e., $I \cup \{L\}$ is not contained in any answer set), it is called on $I \cup \{\texttt{not } L\}$. The classical literal L plays the role of a branching variable of a SAT solver. And indeed the selection of a "good" literal L is crucial for the performance of an ASP system. In the next section, we describe the heuristic criterion adopted by `DLV` for the selection of such branching literals, and how Select is implemented.

4 Evaluation of the Heuristic Function

In this section, we define the heuristic criterion adopted in the `DLV` system and we describe how it is evaluated.

[2] Observe that the interpretations built during the computation are 3-valued and an interpretation I is a set of ground literals. A ground classical literal A is True (resp. False) w.r.t. to I if $A \in I$ (resp. $\texttt{not } A \in I$); otherwise A is Undefined w.r.t. I.

The heuristics of DLV is a "dynamic heuristics" (the ASP equivalent of UP heuristics for SAT), that is, the heuristic value of a literal Q depends on the result of taking Q 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 classical literals; rather, it considers only a subset of the undefined classical literals, called *possibly-true* literals. The correctness of this strategy, adopted since the first release of DLV, is shown in [LRS97].

Definition 1. A *Possibly-True (PT) literal* of $\mathcal{P}$ w.r.t. an interpretation I is an undefined classical literal p such that there exists a rule $r \in Ground(\mathcal{P})$ for which all of the following conditions hold:

1. p is in the head of r: $p \in H(r)$;
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. the negative body of r is not false w.r.t. I: $I \cap \{a : \texttt{not}\ a \in B^-(r)\} = \emptyset$.

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

Example 1. Consider the program $\mathcal{P} = \{a \vee b\ \texttt{:-}\ c.\ \ d\ \texttt{:-}\ \texttt{not}\ a, \texttt{not}\ f.\ \ e \vee f\ \texttt{:-}\ k.\}$ and let $I = \{c\}$ be an interpretation for $\mathcal{P}$, then $PT_{\mathcal{P}}(I) = \{a, b, d\}$.

As shown in Figure 2 (initial **foreach** statement), DLV's heuristic function is evaluated only on the PT literals. It is worthwhile noting, however, that the PT literals do not always restrict the set of classical literals to be looked-ahead, since *all* undefined literals are PT in some cases. For instance, in the program encoding 3SAT (see Section 7.1) 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 a given stage of the computation, the PTs are only 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.

Let us now turn our attention to the heuristic criterion adopted in DLV to choose the "best" among the PT literals.

A peculiar property of answer sets is *supportedness*: For each true classical literal A in an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true literal in the head of r (r is then called a *supporting rule* for A). Since an ASP system must eventually converge to a supported interpretation, ASP systems try to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, the DLV system counts the number of *UnsupportedTrue (UT)* literals, i.e., classical literals which are true in the current interpretation but still miss a supporting rule (in [FLP99] UTs, called MBTs there, are discussed in detail). For instance, the rule :- *not* x implies that x must be true in each answer set of the program, but it does not give a "support" for x. Thus, in the DLV system x is assumed true in the current interpretation to satisfy that rule, and it is added to the set of UnsupportedTrue literals; it will be removed from this set once a supporting rule for x will be found (e.g., $x \vee b$:- c is a supporting rule for x in the interpretation $I = \{x, \texttt{not}\ b, c\}$).

```
Procedure Select(var I: Interpretation, var L: ClassicalLiteral);
var  I_A^+, I_A^-: Interpretation;
begin
   L := NULL;
   foreach A ∈ PT_P(I) do
        I_A^+ := DetCons(I ∪ {A});    (* look-ahead for A *)
        if I_A^+ = L then I := I ∪ {not A};
        else I_A^- := DetCons(I ∪ {not A}); (* look-ahead for not A *)
             if I_A^- = L then I := I ∪ {A}; endif
        endif
        if I_A^+ ≠ L and I_A^- ≠ L then   (* no inconsistency has arisen *)
             if L = NULL then L := A; (* first literal, no comparison *)
               (* compare A against L w.r.t. the heuristic; *)
             elseif ( UT(I_A^+) + UT(I_A^-) ) < ( UT(I_L^+) + UT(I_L^-) ) then L := A;
             elseif ( UT_2(I_A^+) + UT_2(I_A^-) ) < ( UT_2(I_L^+) + UT_2(I_L^-) ) then L := A;
             elseif ( UT_3(I_A^+) + UT_3(I_A^-) ) < ( UT_3(I_L^+) + UT_3(I_L^-) ) then L := A;
             elseif ( US(I_A^+) + US(I_A^-) ) < ( US(I_L^+) + US(I_L^-) ) then L := A;
   endfor
end;
```

Fig. 2. Selection of the Branching Literal by DLV's Heuristic

Intuitively, since the set of UnsupportedTrue literals must eventually be empty when an answer set is reached, the heuristic of DLV tries to minimize the number of UT literals, taking particular care of those UT literals which are more "in danger" (an UT literal appearing in the head of fewer rules is more in danger than a literal appearing in the head of many rules).

Given an interpretation I, let $UT(I)$ be the number of UT literals in I. Moreover, let $UT_2(I)$ and $UT_3(I)$ be the number of UT literals occurring, respectively, in the heads of 2 and 3 rules (which are not already satisfied w.r.t. I, and can therefore be potentially used to support the UT literal).[3] The heuristic of DLV considers $UT(I)$, $UT_2(I)$ and $UT_3(I)$ in a prioritized way to favor literals yielding interpretations with fewer $UT/UT_2/UT_3$ literals (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic minimizes $US(I)$ the number of unsatisfied rules w.r.t. I.

Since the failure of the computation branch selecting A True starts a new branch assuming not A (see last instruction in Figure 1), the heuristic criterion considers the effect of choosing a literal A and its complement not A in a balanced way. To this end, the counters $UT(\mathrm{I}_\mathrm{A}^+)$, $UT_2(\mathrm{I}_\mathrm{A}^+)$, $UT_3(\mathrm{I}_\mathrm{A}^+)$, and $US(\mathrm{I}_\mathrm{A}^+)$, resulting from the look-ahead on $I \cup \{A\}$, are ordinately added to the counters $UT(\mathrm{I}_\mathrm{A}^-)$, $UT_2(\mathrm{I}_\mathrm{A}^-)$, $UT_3(\mathrm{I}_\mathrm{A}^-)$, and $US(\mathrm{I}_\mathrm{A}^-)$, resulting from the look-ahead on $I \cup \{\texttt{not } A\}$, when evaluating the heuristics.

[3] UT_1 literals do not exist in DLV computations. Whenever a rule r is the last potentially supporting rule for an UT literal A, then A is inferred via r (see [FLP99]).

5 Look-Ahead Equivalences

Dynamic heuristics vary only in the interpretation resulting from the function $DetCons(\mathrm{I}_\mathrm{A}^+)$ (resp. $DetCons(\mathrm{I}_\mathrm{A}^-)$). It is therefore interesting to identify cases where two literals L and L' are *look-ahead equivalent*, i.e., $DetCons(I_L) = DetCons(I_{L'})$, since one of the two look-ahead computations could be saved. This notion of equivalence is formalized next.

Definition 2. *Let p and q be two undefined literals w.r.t. an interpretation I. p and q are* look-ahead equivalent *if $DetCons(I \cup \{p\}) = DetCons(I \cup \{q\})$.*

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 $\mathcal{P}$, a classical literal a, and a (3-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: (i) a occurs in the head of r, (ii) no literal in $H(r) - \{a\}$ is true w.r.t. I, and (iii) 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.

We can now formulate the following:

Proposition 1. *If two undefined classical 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 classical literals in rules), then it holds that:*

1. *If $psupp_\mathcal{P}(b, I) = 1$, then a and* `not` *b are look-ahead equivalent.*
2. *If $psupp_\mathcal{P}(a, I) = 1$, then* `not` *$a$ and b are look-ahead equivalent.*

Proof. *(Sketch)* Suppose $psupp_\mathcal{P}(b, I) = 1$. Then r is the only rule in $\mathcal{P}$ which might derive b. Since the body of r is already true in I, such a derivation is performed iff a becomes false. Therefore, $DetCons(I)$ either contains both a and `not` b or it contains none of them. A symmetric argument shows item 2. ∎

Example 2. Consider the program $\{a \vee b.\}$ and $I = \emptyset$. Both a and b are PT literals, so look-ahead for a, `not` a, b, and `not` b is performed, i.e. we compute $DetCons(\{a\}) = \{a, \texttt{not } b\}$, $DetCons(\{\texttt{not } a\}) = \{\texttt{not } a, b\}$, $DetCons(\{b\}) = \{\texttt{not } a, b\}$, and $DetCons(\{\texttt{not } b\}) = \{a, \texttt{not } b\}$. In this example we can save the look-aheads for `not` b and b because of proposition 1, and thus save half of the look-aheads.

In `DLV` computations, we can recognize the applicability of Proposition 1 very efficiently and avoid extraneous look-aheads. Experimental results reported in Section 8 will show that we avoid up to 50% of look-aheads in some cases (e.g. on 3SAT) by exploiting this simple condition.

6 2-Layered Heuristic

In [LA97] a different idea on reducing look-aheads is presented: An easy-to-compute heuristics is defined as a first layer, and look-ahead is only computed on those possible choices which look promising w.r.t. to this easier heuristics. This gives a kind of 2-layered heuristic.

The simple heuristic criteria defined in [LA97] involve the number of *binary clauses* a classical literal occurs in. The rationale is that this is the number of immediate propagations that can be performed during the look-ahead. This idea can be directly transferred to our ASP framework:

Definition 4. *A* binary clause *is a rule which contains exactly two undefined classical literals w.r.t. an interpretation I. The* number of binary occurrences *of an undefined literal a is the number of binary clauses a occurs in.*

Note that this notion directly corresponds to the number of immediate propagations which can be performed by assuming a and `not` a, so it matches the intuition of [LA97]. 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 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., make look-ahead only on these literals).

Note that our first-layer heuristics is inspired by the same intuition as the first-layer heuristics in [LA97], even though it is not precisely the same.

7 Benchmarks

7.1 Benchmark Programs

To evaluate the impact of the two optimization techniques presented in the previous sections, we chose a couple of benchmark problems: 3SAT, Blocksworld Planning, Hamiltonian Path, and Strategic Companies.

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

> Let Φ be a propositional formula in conjunctive normal form (CNF) $\Phi = \bigwedge_{i=1}^{n}(d_{i,1} \vee \ldots \vee d_{i,3})$ where the $d_{i,j}$ are literals over the 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 and can be easily represented in our formalism as follows: For each 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 true: $x_i \vee nx_i$. For each clause $d_1 \vee \ldots \vee d_3$ in Φ we add the constraint `:- not` $\bar{d}_1, \ldots,$ `not` $\bar{d}_3$. where $\bar{d}_i$ ($1 \leq i \leq 3$) is x_j if d_i is a positive literal x_j, and nx_j if d_i is a negative literal $\neg x_j$.

Hamiltonian Path (HAMPATH) is another 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 $arc(X, Y)$[4], and the starting node is specified by the unary predicate $start$ which contains only a single tuple. Then, the following program solves the problem HAMPATH.

```
% Each node has to be reached.
:-node(X), not reached(X).   reached(X):-start(X).   reached(X):-inPath(Y, X).
% Guess whether to take a path or not.
inPath(X, Y) ∨ outPath(X, Y):-reached(X), arc(X, Y).
% At most one incoming/outgoing arc!
:-inPath(X, Y), inPath(X, Z), Y<>Z.  :-inPath(X, Y), inPath(Z, Y), X<>Z.
```

Blocksworld (BW) is a classic problem from the planning domain, and one of the oldest problems in AI:

> Given a table and a number of blocks in a (known) initial state and a desired goal state, try to reach that goal state by moving one block at a time, such that only unoccupied blocks are moved on top of other unoccupied blocks or the table.

Figure 3 shows a simple example that can be solved in three time steps: First we move block c to the table, then block b on top of a, and finally c on top of b. Due to space restrictions we refer to [Erd99,FLMP99] for complete encodings.

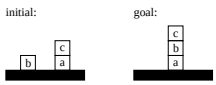

Fig. 3. Simple BW Example

Strategic Companies (STRATCOMP) finally, is a Σ_2^P-complete problem, which has been first described in [CEG97]:

> A holding owns companies $C(1), \ldots, C(c)$, each of which produces some goods. Some of these companies may jointly control another one. This is modeled by means of predicates $prod(P, C1, C2)$ — product P is produced by companies $C1$ and $C2$ — and $contr(C, C1, C2, C3)$ — company C is jointly controlled by $C1$, $C2$ and $C3$.

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

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.

The answer sets of the following natural program correspond one to one to the strategic sets. Checking whether any given company C is strategic is done by brave reasoning: "Is there *any* answer set containing C?"

$$strategic(C1) \vee strategic(C2) \texttt{ :- } prod(P, C1, C2).$$
$$strategic(C) \texttt{ :- } contr(C, C1, C2, C3), strategic(C1), strategic(C2), strategic(C3).$$

As in [CEG97] we assume that each product is produced by at most two companies and each company is jointly controlled by at most three companies.

7.2 Benchmark Data

For 3SAT, we have randomly generated 3CNF formulas over n variables (where n denotes the size as plotted on the x-axis of the graphs in Section 8) using a tool by Selman and Kautz [SK97]. For each size we generated 8 instances, where we kept the ratio between the number of clauses and the number of variables near the cross-over point of 4.3.

The instances for HAMPATH were generated by a tool by Patrik Simons which has been used to compare Smodels against SAT solvers (cf. [Sim00])[5]. Again, for each problem size n we generated 8 instances, always assuming node 1 as the starting node.

The blocksworld problems P3 and P4 have been employed in [Erd99] to compare ASP systems, and can be solved in 8 and 9 steps, respectively. We augmented these by problem P5 which requires 11 steps. For each of these problems, we generated 8 random permutations of the input.

For STRATCOMP, finally, we randomly generated 8 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 *contr* relations per company.

All experiments were performed on a Pentium III/733 machine with 256MB of main memory running GNU/Linux. The binaries were generated with GCC 2.95.2. The input files used for the benchmarks are available on the web at `http://www.dbai.tuwien.ac.at/proj/dlv/lpnmr01.tar.gz`.

8 Experimental Results and Conclusion

The results of our experiments are displayed in the graphs of Figures 4-7. For each problem domain we report two graphs: In both graphs the horizontal axis

[5] available at `http://tcs.hut.fi/Software/smodels/misc/hamilton.tar.gz`

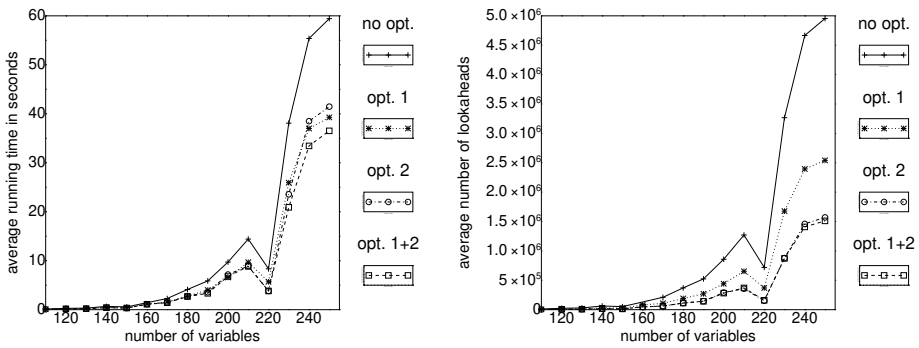

Fig. 4. 3SAT problems, average running times and look-aheads

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 8 instances of the same size we have run (see previous section). The curves labeled by "no opt.", "opt. 1", "opt. 2", and "opt. 1+2", denote, respectively, the initial (unoptimized) version, the look-ahead equivalence optimization, the 2-layered optimization, and the combination of both look-ahead equivalence and 2-layered optimization.

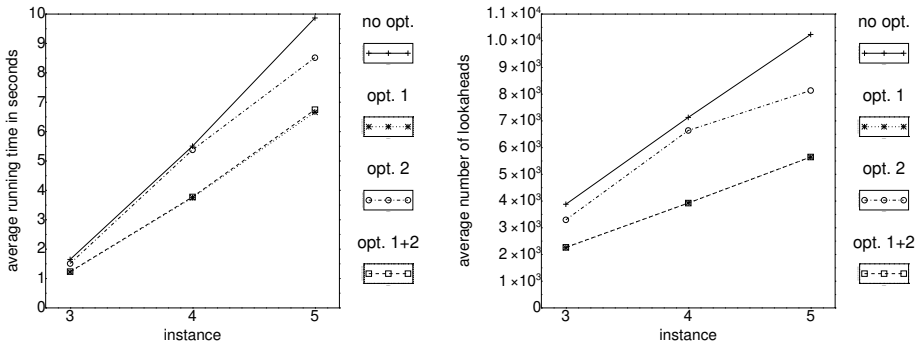

Fig. 5. Blocksworld problems, average running times and look-aheads

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.

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

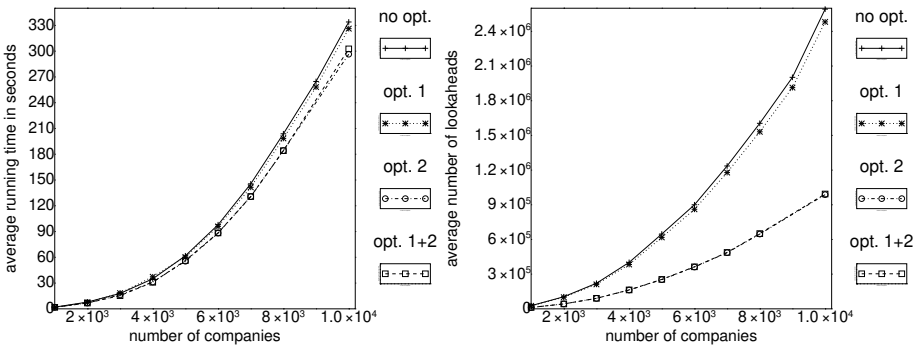

Fig. 6. Strategic Companies, average running times and look-aheads

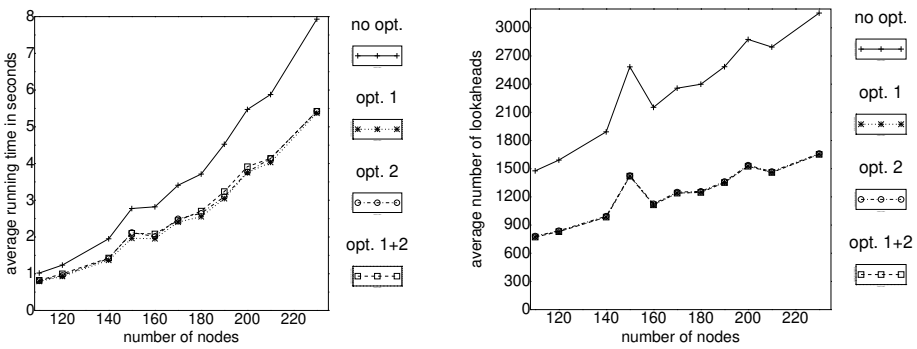

Fig. 7. Hamiltonian Path problems, average running times and look-aheads

The combination of the two optimizations combines the benefits in the sense that performance is always as good as for the best of the two strategies. Indeed, the curve combining the two strategies (opt.1+2) often nearly coincides with the curve of the best of opt.1 and opt.2, e.g. for Blocksworld opt.1+2 and opt.1 are almost equal, while for Strategic Companies opt.1+2 and opt.2 coincide. On Hamiltonian Path opt.1, opt.2, and opt.1+2 all give the same speed-up. Finally, in the case of 3SAT there are even better results for opt.1+2 than for any of the two methods alone.

Note that for opt.2 (and opt.1+2), 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 trees. For opt.1 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.

Thus, both optimizations turned out to be useful, and we have incorporated their combination in the version of `DLV` released in June 2001. We believe that this is a promising way towards the improvement of ASP systems that should be subject of further investigation. Indeed, besides optimizing the implementation

of the techniques proposed in this paper, we have already planned future work to explore other promising ways to reduce the number of look-aheads.

References

[CEG97] M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *IEEE TKDE*, 9(3):448–463, 1997.

[EFLP99] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications*, 12(1–2):99–111, 1999.

[EFLP00] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

[ELM+98a] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. *FQAS'98*, pp. 148–163. Springer.

[ELM+98b] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. *KR'98*, pp. 406–417. Morgan Kaufmann Publishers, 1998.

[Erd99] E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft. `http://www.cs.utexas.edu/users/esra/papers.html`, 1999.

[FLMP99] W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. *DDLP'99*, pp. 135–139.

[FLP99] W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. *LPNMR'99*, pp. 177–191.

[FLP01] W. Faber, N. Leone, and G. Pfeifer. Experimenting with heuristics for answer set programming. In *IJCAI 2001*, Seattle, WA, USA, August 2001. To appear.

[GL91] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[LA97] C.L. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI 1997*, pp. 366–371.

[Lif96] V. Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford, 1996.

[Lif99] V. Lifschitz. Answer set planning. *ICLP'99*, pp. 23–37. The MIT Press.

[LRS97] N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, 1997.

[Min82] J. Minker. On Indefinite Data Bases and the Closed World Assumption. *CADE'82*, LNCS 138, pp. 292–308, New York, 1982. Springer.

[Sim00] P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Finland, 2000.

[SK97] B. Selman and H. Kautz, 1997. `ftp://ftp.research.att.com/dist/ai/`.