Pushing Goal Derivation in DLP Computations*

Wolfgang Faber, Nicola Leone, and Gerald Pfeifer

Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria
{faber,leone,pfeifer}@dbai.tuwien.ac.at

Abstract. dlv is a knowledge representation system, based on disjunctive logic programming, which offers front-ends to several advanced KR formalisms. This paper describes new techniques for the computation of answer sets of disjunctive logic programs, that have been developed and implemented in the dlv system. These techniques try to "push" the query goals in the process of model generation (query goals are often present either explicitly, like in planning and diagnosis, or implicitly in the form of integrity constraints). This way, a lot of useless models are discarded "a priori" and the computation converges rapidly toward the generation of the "right" answer set. A few preliminary benchmarks show dramatic efficiency gains due to the new techniques.
Keywords: *Disjunctive Logic Programming, Algorithms, Heuristics.*

1 Introduction

dlv is a knowledge representation system, based on disjunctive logic programming (DLP) [13, 8], which offers front-ends to several advanced KR formalisms [5, 4, 2]. A strong point of dlv is its highly expressive language, which allows to represent very hard problems (even Σ_2^P-hard problems) in an elegant and natural fashion. An efficient support for such an expressive language requires the use of smart algorithms and optimization techniques, that are able to deal with hard computational tasks. This paper describes new techniques for the computation of Disjunctive Logic Programs, that have been developed and implemented in the dlv system.

We start from the observation that the DLP encoding of problems, and particularly of AI problems, often contains query goals. Frequently, these goals are explicit. In the DLP encoding of planning problems [6, 11], for instance, we usually look for a particular answer set where a query goal, representing the desired evolution of the system, is true. Similarly, in abductive diagnosis we look for answer sets where the observation (encoded as a query goal) is true [2].

Sometimes the query goals are implicitly expressed by integrity constraints [14, 12]. Consider, for instance, the program $\mathcal{P}_{hp}$ in Figure 1, that we will use as

* This work was supported by FWF (Austrian Science Funds) under the projects P11580-MAT and Z29-INF.

```
(i)  reached(X) :- start(X).
(ii) reached(X) :- reached(Y), inPath(Y,X).
(iii) inPath(X,Y) ∨ outPath(X,Y) :- arc(X,Y).
(iv) :- inPath(X,Y), inPath(X,Y1), Y <> Y1.
     :- inPath(X,Y), inPath(X1,Y), X <> X1.
(v)  :- node(X), not reached(X).
```

Fig. 1. The Hamiltonian path program $\mathcal{P}_{hp}$

as a running example throughout the paper. The program computes the *Hamiltonian paths* of a directed graph starting at a given node. (Recall that a Hamiltonian path of a directed graph is a path through that graph that reaches each node exactly once.) In our example, the graph is given by relations $arc(X, Y)$ and $node(X)$ denoting its arcs and nodes, respectively, plus a starting node $start(X)$.

Rule (iii) "guesses" a set S of arcs of the graph. Constraints (iv) enforce a path property on S. Constraint (v) imposes that all nodes are reached by S. The latter constraint is like a query $reached(1), ..., reached(n)?$, where $1..n$ are the nodes of the input graph,[1] asking to compute an answer set in which all these atoms are true. [2]

The techniques proposed in this paper try to "push" query goals in the DLP computation driving it towards the generation of the desired answer sets. The contributions of the paper are the following:

- We define a new truth value, called *must-be-true (mbt)*, for the partial interpretations we use during the computation. Intuitively, the *mbt* truth value is assigned to atoms that cannot be derived from any program rule at the current computation step, but for sure must eventually be true in the answer set to be computed (e.g., in the example above the atom $reached(k)$ is *mbt*, for each node k of the graph). These atoms are not immediately taken as true in order to guarantee the "supportedness" of the interpretation at hand (i.e., that every true atom is derivable from some program rule). Supportedness is an important peculiarity of the logic programming semantics, and ensuring this property is a main difference between SAT checkers and logic programming systems.
- We provide a new function $det\_cons(I)$ which extends an interpretation I at hand in a deterministic way, such that *every* answer set containing I also contains its extension computed by *det_cons*. This function "propagates" *mbt* truth values as much as possible, performing also a sort of backward chaining when it is possible in a deterministic way (e.g., undefined atoms $a_1, ..., a_n$ become *mbt* if b `:-` $a_1, ..., a_n$ is the only rule having the *mbt* b in the head).

[1] Queries are indeed translated into constraints in `dlv`.

[2] Note that the starting node is not strictly required to be the end node of an arc of the Hamiltonian path in this encoding.

– Most importantly, we propose a new heuristic function, driving the computation according with the criterion of "finding support" for the *mbt* atoms (i.e., making them true through a program rule deriving them).
– We implement the above ideas in the `dlv` system, and report some results of experiments. Even if the results are preliminary, and further experiments have to be done, these experimental results undoubtedly evidence the usefulness of the proposed techniques, through a dramatic performance gain on relevant problems.

It is worthwhile noting that the function *det_cons* employs a disjunctive extension of Fitting's operator [7] and extends to the disjunctive case a number of techniques already used in smodels [15] and XSB [1] for the computation of disjunction-free logic programs. The main contribution of the paper is the definition and the experimentation of the heuristics based on the new notion of *mbt*; we are not aware of any similar heuristics for nonmonotonic systems.

2 The Language of `dlv`

An *atom* is an expression of the form $a(t_1, \ldots, t_f)$ where a is the predicate name of the atom and each t_i $(1 \leq i \leq f)$ is either a constant or a variable. A literal is an expression of the form L or `, not` L, where `, not` is the *negation as failure* symbol, and L is an *extended atom*, i.e., either an atom A, or an atom preceded by the *strong negation* symbol '$-$' $(-A)$.

A `dlv` *program* is a set of rules and constraints. A *rule* r has the form
$a_1 \vee \ldots \vee a_m$ `:-` b_1`, ...,` b_k`, not` b_{k+1}`, ..., not` b_n where $m \geq 1$ and $n \geq 0$.

The part to the left of `:-` is called the *head* of the rule, the part to the right is the *body*. We denote by $H(r)$ resp. $B(r)$ the set of literals occurring in the head resp. body of r; $B^+(r)$ (resp. $B^-(r)$) denotes the subset of positive (resp. negative) literals in $B(r)$ (for the rule above we have $H(r) = \{a_1, \ldots, a_m\}$, $B^+(r) = \{b_1, \ldots, b_k\}$, $B^-(r) = \{$`not` $b_{k+1}, \ldots,$ `not` $b_n\}$, and $B(r) = B^+(r) \cup B^-(r)$). A *constraint* is a rule with an empty head ($m = 0$).

Ground queries of the form b_1`, ...,` b_k`, not` b_{k+1}`, ..., not` b_n`?` are allowed and are translated to constraints internally. For example, the query `a, not b?` will generate the two constraints `:- not a.` and `:- b.`

The semantics of a `dlv` program is given by its consistent answer sets [8].

It is worth noting that, in `dlv` computations, integrity constraints are simply treated as rules always having a $false$ head. Moreover, every strongly negated atom $-p(\bar{t})$ is replaced by a new atom $p'(\bar{t})$, where p' is a fresh predicate symbol. For any such new atom $p'(\bar{t})$, a constraint `:-` $p(\bar{t}), p'(\bar{t})$ is added to the program. Thus, though the system takes as input programs written in extended disjunctive DLP, they are treated internally as "traditional" disjunctive DLP programs.

3 Deriving Deterministic Consequences

In this section, we describe the function *det_cons* which, given a ground program $\mathcal{P}$ and a (partial) interpretation I, derives certain knowledge concerning the

literals which are not yet decided in I. In other words, *det_cons* extends the interpretation I in a deterministic way, such that *every* answer set containing I also contains its extension computed by *det_cons*. This function is crucial for the efficiency of the system: the larger the interpretation it derives, the smaller the remaining search space.

During the computation, we deal with four-valued interpretations, where we consider the following truth values: *true (T), must-be-true (M* or *mbt), undefined (U),* and *false (F).* Negation acts as follows on these truth values: `not` $T = F$, `not` $M = F$, `not` $U = U$, and `not` $F = T$. We associate the total ordering $T > M > U > F$ to the truth values.

An interpretation I for $\mathcal{P}$ is a total mapping from $B_\mathcal{P}$ to $\{T, M, U, F\}$. We denote by I^T (resp. I^M, I^U, I^F) the set of atoms whose truth value according to I is *true* (resp. *must-be-true, undefined, false*). Intuitively, the *mbt* truth value is assigned to atoms that cannot be derived from any program rule at the current computation step, but must eventually be true in the answer set to be computed. These atoms are not immediately taken as true in order to guarantee the "supportedness" of the interpretation at hand. This is a main peculiarity of answer sets w.r.t. ordinary models: Any atom p belonging to an answer set of $\mathcal{P}$ has a rule which *supports* p. Formally, p is *true* w.r.t. a given answer set S if and only if there exists a rule $r \in \mathcal{P}$ such that $p \in H(r)$, $H(r) - \{p\}$ is $false$ w.r.t. S, and $B(r)$ is *true* w.r.t. $S - \{p\}$. Thus, enforcing supportedness is a principal difference between DLP systems and satisfiability solvers (like, e.g., the Davis-Putnam procedure).

Given an interpretation I for $\mathcal{P}$, the function val_I is defined as follows: For any atom $p \in B_\mathcal{P}$, $val_I(p) = I(p)$, and $val_I(\texttt{not}\ p) = \texttt{not}\ val_I(p)$. Accordingly, for a ground rule $r \in \mathcal{P}$, we define $val_I(H(r))$ (resp., $val_I(B(r))$) as the maximum (minimum) value assigned by val_I over the literals in $H(r)$ ($B(r)$). If $H(r) = \emptyset$, i.e., r is a constraint, then $val_I(H(r)) = F$. r is *satisfied* w.r.t. I if the truth value of its head is not less than the truth value of its body, i.e. $val_I(H(r)) \geq val_I(B(r))$. Moreover, for every atom p, we define $support(p)$ as the set of rules in $ground(\mathcal{P})$ such that $val_I(H(r) - \{p\}) < M$ and $val_I(B(r)) > F$, i.e., the rules which can be potentially used to derive the truth of p "starting" from the interpretation I. $|support(p)|$ denotes the cardinality of $support(p)$.

The procedure *det_cons* is shown in Figure 2. Given an interpretation I for $\mathcal{P}$, it extends I by what we call the *deterministic consequences* of I w.r.t. $\mathcal{P}$. It can assign F, M or T to any undefined atom, but can only assign T to *mbt* atoms. *det_cons* also detects inconsistencies, e.g. if some *mbt* atom should get the value F.

As long as *det_cons* has modified the interpretation I for the ground program $\mathcal{P}$, the Boolean variable $modified$ is true at the end of the **repeat** loop of Step 2. If any inconsistency is detected, the procedure immediately aborts by means of an **exit** instruction.

Steps 4–12 focus on rules which are not satisfied (in the 4-valued sense described above). If the head of the rule is *false* and its body is either *true* or *mbt*, then the procedure exit returning *contradiction* = *true*, because there is no way

```
Procedure det_cons(P: Program; var I: Interpretation; var contradiction: Boolean)
(* Computes the deterministic consequences for P w.r.t. I *)
var modified: Boolean;
begin
(1)     contradiction := false;
(2)     repeat
(3)         modified := false;
            (* Enforce satisfaction of all rules *)
(4)         for each rule r ∈ ground(P) not satisfied w.r.t. I do
(5)             if val_I(B(r)) ≥ M and val_I(H(r)) = F
(6)                 contradiction := true; exit procedure;
(7)             else if val_I(B(r)) ≥ M
                        and val_I(H(r) − {p}) = F for a p ∈ H(r) then
(8)                 I(p) := val_I(B(r)); modified := true;
(9)             if val_I(H(r)) = F and val_I(B(r) − {L}) ≥ M
                for some undefined literal L ∈ B(r) then
(10)                modified := true;
(11)                if L is a positive literal p then I(p) := F;
(12)                else (* L is a negative literal not p *) I(p) := M;
        end for;
        (* Ensure supportedness *)
(13)        if |support(p)| = 0 and I(p) ≥ M for some atom p then
(14)            contradiction := true; exit procedure;
(15)        for each atom p s.t. I(p) = U and |support(p)| = 0 do
(16)            I(p) := F;
(17)        for each atom p s.t. I(p) ≥ M and |support(p)| = 1 do
              Let r be the (unique) rule in support(p);
(18)            for each undefined atom q ∈ (H(r) − {p}) do
(19)                I(q) := F; modified := true;
(20)            for each undefined positive literal q ∈ B(r) do
(21)                I(q) := M; modified := true;
(22)            for each undefined negative literal not q ∈ B(r) do
(23)                I(q) := F; modified := true;
              end for;
(24)    until not modified
end_procedure
```

Fig. 2. Function for computing the deterministic consequences

to satisfy r (recall that *true* and $false$ atoms cannot be changed and *mbt* can evolve only into *true*). Steps 7–12 enforce the satisfaction of a rule $r \in \mathcal{P}$ if this can be done deterministically, i.e., by changing the value of exactly one literal occurring in r. Consider Steps 7–8: If the truth value of $B(r)$ according to I is X, where X is at least M, i.e., *mbt* or *true*, and every atom in the head of r is $false$, except for one atom p, we can draw a deterministic consequence. We enforce the satisfaction of r by incrementing the truth value of p up to the value of $B(r)$.

For instance, if p is either undefined or *mbt* w.r.t. I and $val_I(B(r)) = T$, then I is modified by assigning the value T to p, denoted by $I(p) := T$ in the algorithm. Note that this is the only step which can assign the value *true* to an atom, that is, *det_cons* assigns the value *true* to an atom p only if p is "supported".

Now, consider Steps 9–12: If the head of r is $false$, but its body is at least *mbt*, except for one undefined literal L, then L should get the truth value $false$ in order to satisfy r. Note that, if L is a negative literal `not` p, this is accomplished by setting $I(p) := M$. Indeed, declaring p *true* would not guarantee the "supportedness" of this atom.

Steps 13–23 draw deterministic conclusions following the "supportedness" principle; they are a main novelty of our approach. In particular, if a *true* or *mbt* atom has no supporting rule according to the interpretation I, then we get a contradiction (Step 14), while an undefined atom without any supporting rule can be declared $false$ (Steps 15–16).

If a *true* or *mbt* atom p has only one supporting rule r, i.e., $support(p) = \{r\}$, then r must be able to derive the truth of p. Thus, we enforce that p is derivable from r assigning suitable truth values to every undefined literal occurring in r (see Steps 18–23). This is a sort of backward propagation step: From the truth of the head we derive that all body literals must be true.

Example 1. Consider the program from Figure 1 applied to (the encoding of) the graph of Figure 3 starting with the "empty" interpretation I, where $I^T = I^M = I^F = \emptyset$ and $I^U = B_{\mathcal{P}}$.

By rule (i), `reached(a)` is immediately derived (by steps 4–8). The constraint (v) essentially serves as a query that assures that all nodes are indeed reached. As `node(X)` is true for all nodes, `reached(X)` is derived as *mbt* by means of Steps 9–12, for each $X \in \{a, b, c, d, e\}$.

The *mbt* atom `reached(b)` is only derivable by a single ground instance[3] of rule (ii), namely `reached(b) :- reached(a), inPath(a,b).` At this point, the backward propagation step described above comes into play and sets `inPath(a,b)` to *mbt* (lines 17–21). Then, *support(`outPath(a,b)`)* becomes empty, since the only rule with `outPath(a,b)` in the head contains also the *mbt* `inPath(a,b)`. Thus, `outPath(a,b)` is derived as *false* (lines 15–16). In turn, this causes Steps 4–8 to derive `inPath(a,b)` as *true*. Now we easily derive `reached(b)` as true from (ii).

Also `inPath(c,d)` and `outPath(c,d)` are derived as *true* and *false*, respectively, in analogy to `inPath(a,b)` and `outPath(a,b)`.

Each node in a Hamiltonian path has exactly one outgoing arc, and indeed Steps 9–12 derive `inPath(a,c)` and `inPath(a,e)` as *false*, which then leads to `outPath(a,c)` and `outPath(a,e)` to be set to *true*.

Now we derive `inPath(b,c)` and `inPath(d,e)` as *true* in the same way we derived `inPath(a,b)` above, and eventually are able to obtain `reached(c)`, `reached(d)`, and `reached(e)`. That is, starting from an "empty" interpreta-

[3] Note that the instantiation procedure of `dlv` generates only ground rules that are constructible from the facts in the input ([4]).

tion, a single invocation of *det_cons* has deterministically and efficiently found the Hamiltonian path for this graph. ∎

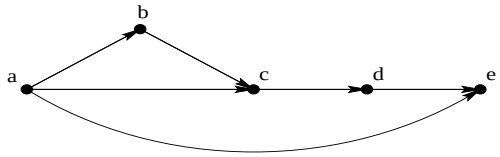


Fig. 3. Example graph 1

Recall that a total model S for $\mathcal{P}$ is an interpretation satisfying the following properties: (i) every rule of $\mathcal{P}$ is satisfied w.r.t. S, and (ii) $S^M \cup S^U = \emptyset$, i.e., every atom is either true or false w.r.t. S.

The partial order $\preceq$ on the four truth values is defined through the following relationships: $U \preceq F$, $U \preceq M$, $U \preceq T$, and $M \preceq T$; moreover, $X \preceq X$ for any $X \in \{F, U, M, T\}$. We say that an interpretation I' extends an interpretation I if, for each atom p, $I(p) \preceq I'(p)$. Intuitively, I' represents more concrete knowledge than I does.

The correctness of our algorithm relies on the following property of *det_cons*.

Theorem 1. *Let $\mathcal{P}$ and I be the program and the interpretation, resp., given as inputs to* det_cons*, and denote by I' and contradiction$'$ the value of the variables I and contradiction, resp., at the end of the procedure. Then,*

1. *I' extends I;*
2. *every answer set S for $\mathcal{P}$ which extends I, extends I' as well;*
3. *if contradiction$'$ holds, then no answer set S' for $\mathcal{P}$ extends I.* ∎

It is worthwhile noting that *det_cons* has been implemented very carefully in our system. By using sophisticated data structures for representing rules and interpretations, it runs in linear time, i.e., in time $O(\| \mathcal{P} \| + \| I \|)$, where $\| \cdot \|$ denotes the size of an object.

4 Overall Model Generation Algorithm

In this section we briefly review the model generation algorithm (*Model Generator*) of the `dlv` system, in order to show how the new notion of *mbt* and the new function *det_cons* are employed.

The Model Generator (MG) produces a set of interpretations that are "candidates" for answer sets, which are then submitted to the Model Checker for verification. The Model Generator essentially relies on a backtracking technique which spans the search space for computing all answer sets.

Basically, the MG works as follows: (1) Derive what is deterministically derivable from the program, (2) make an "educated" guess for one of those literals

which have not been decided yet, and (3) propagate the consequences of this choice. This process is recursively applied until either a contradiction arises, or no further guess can be made. In the former case, MG backtracks and modifies the last choice; in the latter case, we have an answer set candidate and the Model Checker is called. If the candidate is not an answer set, backtracking is performed.

To formalize what we have called "educated guess" before, we introduce the concept of a possibly-true (PT) literal:

Definition 1. Let I be an interpretation for $\mathcal{P}$.

A *positive PT literal* of $\mathcal{P}$ w.r.t. I is a positive literal p such that $U \leq I(p) \leq M$ and there exists a rule $r \in ground(\mathcal{P})$ for which all of the following conditions hold:

1. $p \in H(r)$;
2. $val_I(H(r)) < T$ (i.e., the head is not true w.r.t. I);
3. $val_I(B(r)) = T$ (i.e., the body is true w.r.t. I).

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

1. `not` $q \in B^-(r)$;
2. $val_I(H(r)) < T$ (i.e., the head is not true w.r.t. I);
3. $val_I(B^+(r)) = T$ (i.e., the body is true w.r.t. I).
4. $val_I(B^-(r)) \geq U$ (i.e., no negative literal of the body is false w.r.t. I).

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

Example 2. Consider the program $\mathcal{P} = \{a \vee b$ `:-` $c,$ `not` $d.\ \ e$ `:-` $c,$ `not` $f.\}$ and let $I = \{c,$ `not` $d\}$ be an interpretation for $\mathcal{P}$. Then, we have three PT literals of $\mathcal{P}$ w.r.t. I: a, b and `not` f. ∎

The actual algorithm for computing answer sets is shown in Figure 4. There, *isAnswerSet* is a function which returns true iff I^T is an answer set for $\mathcal{P}$. It is worth noting that the essence of the MG, based on the notion of PT, has not significantly changed w.r.t. previous versions; the reader is referred to [10, 3, 4] for further details on the other features.

5 Heuristics

In this section we focus on the question how to select *PT literals* in line (6) of *ComputeAnswerSets* in Figure 4, such that the likelihood of finding an answer set is maximized.

To this end we employ so-called "lookahead", that is, we temporarily assume the truth of one PT literal at a time [4] and perform the deterministic derivations,

[4] For a negative literal this means assigning false to its atom.

Algorithm ComputeAnswerSets
Input: A ground DLP program $\mathcal{P}$.
Output: The answer sets of $\mathcal{P}$ (if any).

```
Procedure ComputeAnswerSets(I: Interpretation)
(* The procedure outputs all answer sets of P *)
var  Q: SetOfLiterals; L: Literal;
(1)      det_cons(P, I,contradiction);
(2)      if contradiction then  exit procedure;
(3)      if (PT_P(I) = ∅) then (* I^T ∪ I^M is a model of P *)
(4)           if (I^M = ∅) and isAnswerSet(P, I^T) then
(5)                output I^T; (* I^T is an answer set *)
         else
(6)           Take a literal L from PT_P(I);
              (* Assume the truth of a PT literal *)
(7)           if L is a negative literal not p then
(8)                I(p) := F;
              else (* L is a positive literal *)
(9)                I(L) := T;
(10)          ComputeAnswerSets(I);
              (* At this point all answer sets containing I ∪ {L} have been generated *)
              (* L must be false in following computations *)
(11)          if L is a negative literal not p then
(12)               I(p) := M;
              else
(13)               I(L) = F;
(14)          ComputeAnswerSets(I);
end_procedure

var I: Interpretation;
begin   (* Main *)
     I^T := ∅;  I^F := ∅;  I^M := ∅;  I^U := B_P;
     ComputeAnswerSets(I);
end.
```

Fig. 4. Algorithm for the Computation of Answer Sets

i.e. we apply the *det_cons* function. On the basis of the changes which have been derived during this lookahead, we then make the decision which PT literal should be taken. (Note that the *smodels* system [15, 16] also employs lookahead, but they use a completely different heuristics.)

Definition 2. A *mbt* atom p is said to be of level n (w.r.t. an interpretation I), if $|support(p)| = n$ (w.r.t. I).

During the lookahead we record the following counters for each PT literal p:

$mbt^-(p)$ The overall number of eliminated *mbt* atoms (*mbt* which became true).

$mbt^+(p)$ The overall number of inserted mbt atoms (*undefined* which became mbt).
$mbt_2^-(p)$ The number of eliminated mbt atoms of level 2.
$mbt_2^+(p)$ The number of inserted mbt atoms of level 2.
$mbt_3^-(p)$ The number of eliminated mbt atoms of level 3.
$mbt_3^+(p)$ The number of inserted mbt atoms of level 3.

The respective level is w.r.t. the interpretation at the moment the mbt atom is assigned true.

In addition, we define some difference functions:

$$\begin{aligned}\Delta_{mbt}(p) &= mbt^-(p) - mbt^+(p)\\ \Delta_{mbt2}(p) &= mbt_2^-(p) - mbt_2^+(p)\\ \Delta_{mbt3}(p) &= mbt_3^-(p) - mbt_3^+(p)\end{aligned}$$ ∎

Concerning heuristics itself, we have defined a heuristic relation over the set of PT literals as follows:

Definition 3. Given two PT literals `a` and `b`, we define an ordering relation $>$ as follows:

If $(mbt^-(a) = 0 \wedge mbt^-(b) > 0) \vee \ (mbt^-(a) > 0 \wedge mbt^-(b) = 0)$ then
$a > b \Leftrightarrow mbt^-(a) > mbt^-(b)$
otherwise
$a > b$ holds if one of the following conditions applies:
1. $\Delta_{mbt}(a) > \Delta_{mbt}(b)$
2. $\Delta_{mbt2}(a) > \Delta_{mbt2}(b) \wedge \Delta_{mbt}(a) = \Delta_{mbt}(b)$
3. $\Delta_{mbt3}(a) > \Delta_{mbt3}(b) \wedge \Delta_{mbt}(a) = \Delta_{mbt}(b) \wedge \Delta_{mbt2}(a) = \Delta_{mbt2}(b)$

Further, let $a = b$ be true if $a \not> b \wedge b \not> a$. ∎

In other words, if exactly one of $mbt^-(a)$ and $mbt^-(b)$ is zero, we prefer the PT literal for which mbt^- is non-zero. Otherwise (i.e., both $mbt^-(a)$ and $mbt^-(b)$ are zero or both are non-zero), we prefer the one for which the overall number of mbt atoms becomes smaller. If this number is equal, we prefer the one for which the overall number of mbt atoms of level 2 becomes smaller. If also this number is equal, we use the number of mbt atoms of level 3. Otherwise, we consider them to be equal.

The reasoning behind this relation is that the total number of mbt atoms can be viewed as constraints which are not yet satisfied but eventually have to be for any answer set. So the fewer mbt atoms there are, the smaller is the distance to an answer set. Additionally, mbt atoms of level 2 and 3 are the ones which are the "hardest" to become satisfied (observe that mbt atoms of level 1 are always derived by *det_cons*).

The purpose of the test whether exactly one of $mbt^-(a)$ or $mbt^-(b)$ is zero is that in this case we want to avoid preferring a PT literal, which only introduces new mbt atoms but does not eliminate any, over one which eliminates some but introduces more (the former is like a "null action").

The guessing step in the Model Generator (line (6) in *ComputeAnswerSets* in Figure 4) takes a PT literal which is a maximum w.r.t. $\geq$.

Example 3. Consider again the program for computing Hamiltonian paths shown in Figure 1, now together with the encoding of the graph depicted in Figure 5 plus `start(a)`.

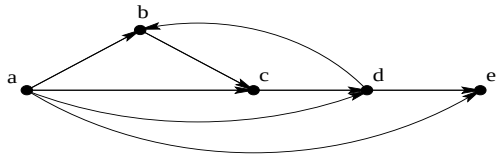


Fig. 5. Example graph 2

By the first call to *det_cons*, only `reached(a)` is set to true, while `reached(b)`, `reached(c)`, `reached(d)`, and `reached(e)` are assigned *mbt* because of the single literal constraints obtained by (v) (see Appendix A).

The choice rule (iii) is instantiated with the arcs (see Appendix A); these rules supply the PT literals (all of which are positive).

Note that the rules which define the predicate `reached` are instantiated in a way such that `reached(n)` occurs in the head of exactly two rules for each node `n` (apart from `a`). This is because each of these nodes has exactly two incoming arcs.

Each of the `reached(n)` (`n` $= \{$`b`$, \ldots,$ `e`$\}$) needs support, but it is not yet known which of the two rules will supply it eventually.

To evaluate the heuristic relation, we perform a lookahead: for each PT L, we assume L true, compute its deterministic consequences (by a call to *det_cons*), and store the values of the respective *mbt* counters.

Let us first consider the PT literal `inPath(a,b)`: Upon assuming it true, we immediately derive `reached(b)` as true, and thus eliminate a *mbt* atom of level 2 (since it occurs in the head of two unsatisfied rules). By statements (9) – (11) in *det_cons* we derive falsity for `inPath(a,c)`, `inPath(a,d)`, `inPath(a,e)`, and `inPath(d,b)`, reflecting the fact that no two arcs in the Hamiltonian path may begin in the same node or end in the same node (constraints (iv) in Figure 1).

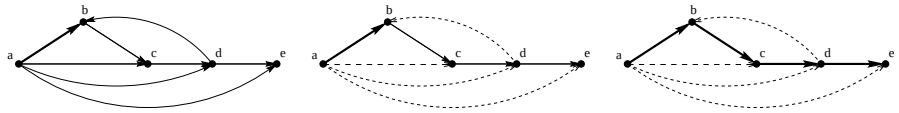


Fig. 6. Steps during lookahead for `inPath(a,b)`

After that, for each of `reached(c)`, `reached(d)`, `reached(e)` only one supporting rule is left, so we can infer that the yet undefined positive body literals of these rules (`inPath(b,c)`, `inPath(c,d)`, `inPath(d,e)`) are *mbt*. Moreover,

since each of them occurs in the head of exactly one rule and the body of this rule is true, we infer them as true immediately afterwards and eventually we also infer `reached(c)`, `reached(d)`, and `reached(e)` as true. These steps are visualized in Figure 6, where bold arcs are in the Hamiltonian path, while dashed arcs are not.

In total, the deterministic derivation has generated 3 new *mbt* atoms (all of which have subsequently been derived as true) and eliminated 7 *mbt* atoms, one of which was of level 2.

All PT literals and their corresponding heuristic-relevant function values, ordered by $\geq$, are shown in Table 1. Those which are not listed (`inPath(a,c)`, `inPath(a,d)`, `outPath(b,c)`, `outPath(c,d)`) generate an inconsistency during propagation.

PT literal	mbt^-	mbt^+	mbt^-_2	mbt^+_2	mbt^-_3	mbt^+_3
`inPath(a,b)`	7	3	1	0	0	0
`outPath(a,e)`	8	4	0	0	0	0
`outPath(d,b)`	8	4	0	0	0	0
`inPath(d,e)`	7	3	0	0	0	0
`inPath(a,e)`	4	3	1	0	0	0
`outPath(a,b)`	5	4	0	0	0	0
`inPath(d,b)`	4	3	0	0	0	0
`outPath(a,b)`	5	4	0	0	0	0
`outPath(a,c)`	1	1	0	0	0	0
`outPath(a,d)`	1	1	0	0	0	0
`inPath(b,c)`	0	0	0	0	0	0
`inPath(c,d)`	0	0	0	0	0	0

Table 1. PT literals and their values, ordered by $\geq$

Thus, following the heuristics, the PT `inPath(a,b)` is chosen by our computation. Then, the propagation of it, done by *det_cons*, immediately leads to the computation of the Hamiltonian path. Thanks to the heuristics only one choice was sufficient! ∎

Note that performing lookahead has an additional merit: If an inconsistency is detected during the propagation of the PT literal, we can then set it to false and apply *det_cons*, thus pruning the search tree quite a bit.

6 Some Experimental Results

We have conducted a number of experiments, in order to show the usefulness of the various techniques introduced in this paper. To this end we have compared several versions of `dlv`:

The first one is the release of February 10^{th}, 1999. It contains only a small part of *det_cons*, notably without the notion of *mbt* atoms and also without the part ensuring supportedness. Also heuristics are not included.

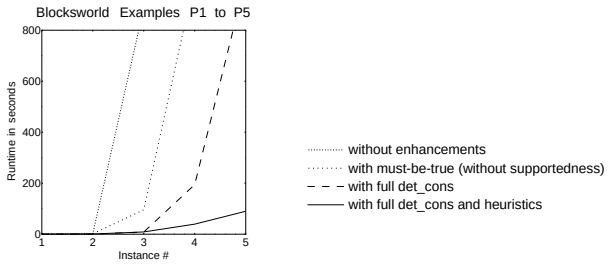
The second one is the release of April 6^{th}, 1999. This one contains the fully implemented first part of *det_cons*, i.e. statements (4) – (12). The part ensuring

supportedness is missing (as in the previous version), and also heuristics are not yet included.

The third version is the release of May 28^{th}, 1999. It contains the full implementation of *det_cons*, but heuristics are not included.

Finally, the fourth version is the previous one, enriched by heuristics. The public release of this version is dated June 8^{th}, 1999.

We have benchmarked a set of blocksworld instances, most of which are taken from [6] (except for P5). We use an encoding of the problem domain which is different from the one in [6], but which is also derived from an encoding in an action language. You can find the domain encoding and the instances in Appendix B.



Since we chose the Hamiltonian path problem as a running example, we have also picked a random graph with 25 nodes and 60 arcs[5] and run the program of Figure 1 and an arbitrarily picked starting node (node 0) with it.

Version 1 could not find a Hamiltonian path within 1000 seconds, while version 2 found one in 716 seconds, and version 3 took 750 seconds. With heuristics enabled, `dlv` was able to find a path in 12.7 seconds!

References

1. W. Chen and D. S. Warren. Computation of Stable Models and Its Integration with Logical Query Processing. *IEEE Transactions on Knowledge and Data Engineering*, 8(5):742–757, 1996.
2. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the `dlv` System. *AI Communications – The European Journal on Artificial Intelligence*, 12(1–2):99–111, 1999.
3. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Nonmonotonic Reasoning. In *Proc. LPNMR '97*, pages 363–374.
4. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. Progress Report on the Disjunctive Deductive Database System `dlv`. In *Proc. FQAS '98*, pages 145–160.
5. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR System `dlv`: Progress Report, Comparisons and Benchmarks. In *Proc. KR'98*, pages 406–417.
6. E. Erdem. Applications of Logic Programming to Planning: Computational Experiments. Unpublished draft, 1999.

[5] Generated by the Stanford Graphbase [9] using `random_graph(25,60,0,0,0,0,0,1,1,60)`.

7. M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
8. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
9. D. E. Knuth. *The Stanford GraphBase : a platform for combinatorial computing.* ACM Press, New York, 1994.
10. N. Leone, P. Rullo, and F. Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics and computation. *Information and Computation*, 135(2):69–112, June 1997.
11. V. Lifschitz. Action Languages, Answer Sets and Planning. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm - A 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.
12. V. W. Marek and M. Truszczyński. Stable Models and an Alternative Logic Programming Paradigm. In K. Apt, V. W. Marek, M. Truszczyński, and D. S. Warren, editors, *The Logic Programming Paradigm - A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.
13. J. Minker. On Indefinite Data Bases and the Closed World Assumption. In *Proc. CADE '82*, pages 292–308.
14. I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, May 1998.
15. I. Niemelä and P. Simons. Smodels - an implementation of the stable model and well-founded semantics for normal logic programs. In *Proc. LPNMR '97*, pages 420–429.
16. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research Report A47, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Finland.

A Instantiation of the Hamiltonian Path Program

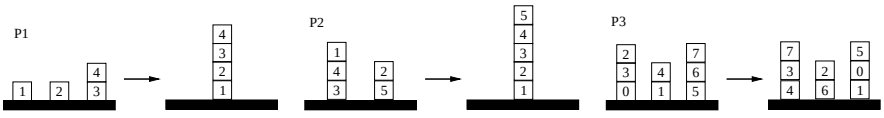
Here are the rules of the Hamiltonian path program in Figure 1, instantiated with example graph 2 of Figure 5:

```
                                :- inPath(a,b), inPath(d,b).
:- inPath(a,b), inPath(a,c).    :- inPath(d,b), inPath(a,b).
:- inPath(a,b), inPath(a,d).    :- inPath(a,c), inPath(b,c).
:- inPath(a,b), inPath(a,e).    :- inPath(b,c), inPath(a,c).
:- inPath(a,c), inPath(a,b).    :- inPath(c,d), inPath(a,d).
:- inPath(a,c), inPath(a,d).    :- inPath(a,d), inPath(c,d).
:- inPath(a,c), inPath(a,e).    :- inPath(d,e), inPath(a,e).
:- inPath(a,d), inPath(a,b).    :- inPath(a,e), inPath(d,e).
:- inPath(a,d), inPath(a,c).
:- inPath(a,d), inPath(a,e).    :- not reached(b).
:- inPath(d,b), inPath(d,e).    :- not reached(c).
:- inPath(d,e), inPath(d,b).    :- not reached(d).
                                :- not reached(e).
```

```
inPath(a,b) ∨ outPath(a,b). reached(b) :- reached(a), inPath(a,b).
inPath(a,c) ∨ outPath(a,c). reached(b) :- reached(d), inPath(d,b).
inPath(a,d) ∨ outPath(a,d). reached(c) :- reached(a), inPath(a,c).
inPath(a,e) ∨ outPath(a,e). reached(c) :- reached(b), inPath(b,c).
inPath(b,c) ∨ outPath(b,c). reached(d) :- reached(c), inPath(c,d).
inPath(c,d) ∨ outPath(c,d). reached(d) :- reached(a), inPath(a,d).
inPath(d,b) ∨ outPath(d,b). reached(e) :- reached(d), inPath(d,e).
inPath(d,e) ∨ outPath(d,e). reached(e) :- reached(a), inPath(a,e).
```

B The Blocksworld Domain and Instances

```
% specification of the move action
move(B,L,T) v -move(B,L,T) :- block(B), location(L), actiontime(T), B <> L.

% the effects of moving a block
on(B,L,T1) :- move(B,L,T), #succ(T,T1).
-on(B,L,T1) :- move(B,_,T), on(B,L,T), #succ(T,T1).

% move preconditions
% a block can be moved only when it's clear
:- move(B,L,T), on(B1,B,T).
% if a block is moved onto another block, the latter must be clear
:- move(B,B1,T), on(B2,B1,T), block(B1).

% concurrent actions are not allowed
:- move(B,_,T), move(B1,_,T), B <> B1.
:- move(_,L,T), move(_,L1,T), L <> L1.

% inertia
on(B,L,T1) :- on(B,L,T), not -on(B,L,T1), #succ(T,T1).

% time at which actions can be initiated
actiontime(T) :- T < #maxint, #int(T).

% location definition (blocks are defined in the problem instances)
true.
location(t) :- true.
location(B) :- block(B).
```



Problem	blocks	steps
P1	4	4
P2	5	6
P3	8	8
P4	11	9
P5	11	11