Computing definite logic programs by partial instantiation*

Vadim Kagan
Department of Computer Science, University of Maryland, College Park, MD 20742, USA

Anil Nerode
Department of Mathematics and Mathematical Sciences Institute, Cornell University, Ithaca, NY 14853, USA

V.S. Subrahmanian
Department of Computer Science, Institute for Advanced Computer Studies and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA

Communicated by D. van Dalen
Received 15 April 1993
Revised 9 August 1993

Abstract

Query processing in ground definite deductive databases is known to correspond precisely to a linear programming problem. However, the “groundedness” requirement is a huge drawback to using linear programming techniques for logic program computations because the ground version of a logic program can be very large when compared to the original logic program. Furthermore, when we move from propositional logic programs to first-order logic programs, this effectively means that functions symbols may not occur in clauses. In this paper, we develop a theory of “Instantiate-by-need” that performs instantiations (not necessarily ground instantiations) only when needed. We prove that this method is sound and complete when computing answer substitutions for non-ground logic programs including those containing function symbols. More importantly, when taken in conjunction with Colmerauer’s result that unification can be viewed as linear programming, this means that resolution with unification can be completely replaced by linear programming as an operational paradigm. Additionally, our tree construction method is not rigidly tied to the linear programming paradigm – we will show that given any method M (which some implementors may prefer) that can compute the set of atomic logical consequences of a propositional logic program, our method can use M to compute (in a sense made precise in the paper), the set of all (not necessarily ground) atoms that are consequences of a first-order logic program.

Correspondence to: V.S. Subrahmanian, Department of Computer Science, Institute for Advanced Computer Studies and Institute for Systems Research, University of Maryland, College Park, MD 20742, USA.
Email: vs@cs.umd.edu

* This work was supported by the Army Research Office under grant numbers DAAL-03-91-C-0027 and DAAL-03-92-G-0225, by the Air Force Office of Scientific Research under grant number F49620-93-1-0065 as well as by the National Science Foundation under Grant number IRI-9109755.

0168-0072/94/$07.00 © 1994 Elsevier Science B.V. All rights reserved
SSDI 0168-0072(93)E0065-V
1. Introduction

Boole [1] showed that translating propositional logic statements of the form \( p \lor q \) to the real linear inequality \( p + q \geq 1 \), translating \( \neg p \) to \( 1 - p \), and using Fourier elimination to reduce the corresponding system of linear inequalities to solved form preserved the following property: "the original propositional logic formula can be satisfied by a propositional truth valuation iff there is a zero–one valued solution to the corresponding inequalities." Nerode suggested to his former student Jeroslow, a 1968 Ph.D. in logic under Nerode, who was a leading specialist in integer linear programming, that he should therefore investigate the feasibility of automated theorem proving via the simplex method and Gomory's integer linear programming method. In 1985, Jeroslow and his student Wang [11] proved the beautiful and simple result that the real linear program associated with a propositional definite logic program and the negation of a query has a real (as opposed to integer) solution if the logic program and the negation of the query is satisfiable by a truth valuation. Subsequently Bell, Nerode, Ng, and Subrahmanian in the Cornell-Maryland LOPS ("Logic and Optimization for Problem Solving") project [2, 3, 4, 5, 17] extended these ideas to nonmonotonic reasoning systems such as general logic programs, stable model semantics, answer set semantics, predicate, parallel and prioritized circumscription, default logics, truth maintenance systems, etc., using mixed integer linear programming.

Specifically, the existence of a model (say a stable model for a propositional general logic program or an extension of a default theory) is equivalent to the existence of a solution for each of a finite sequence of correlated integer linear programs. This applied also to existence of minimal models, answer sets, and circumscriptive fringes of propositional deductive databases. Even though the compiler implementations produced compute the stable models of substantial propositional general logic programs, the work had the limitation that, just as for Jeroslow–Wang, the only way known to handle predicate general logic programs was to "ground out", reducing to the propositional case. As far as we know, until the present paper, Jeroslow and succeeding authors have not removed this limitation. All works to date that we are aware of assume that databases are grounded out prior to linear programming computation. This is a huge drawback for two reasons:

- first, the ground version of a logic program can be very large when compared to the logic program itself, and
- second, the grounding strategy cannot be applied to databases containing function symbols as this may lead to an infinite number of ground clauses.

We have found that a clear understanding of Herbrand's theorem and unification, leads to complete correct proof procedures for (say) definite predicate logic programs which combine real linear programming with unification and "partial substitutions". It is noteworthy that when Jeroslow came to predicate logic in his monograph, he abandoned linear programming and used logical methods, examining decision
procedures for classes of formulas without function symbols that use partial substitutions, that is, non-ground substitutions. But he did not identify partial substitutions as a tool compatible with the use of linear programming as the propositional proof procedure.

In this paper, we develop a theory of partial instantiation for logic program computations that solves the above two problems, and instantiates clauses only on a “need-to-instantiate” basis. We prove that our method yields a sound and complete mechanism for evaluating conjunctive queries to logic programs.

It has been known since Herbrand (1928), or possibly Lowenheim (1915), how to compute, from each predicate logic statement, a sequence of propositional logic statements such that the predicate logic statement is unsatisfiable if and only if one of the sequence of propositional logic statements is unsatisfiable. Following Boole, and using integer programming as a proof procedure, this corresponds to solving a sequence of integer programming problems. The sequence corresponding to query evaluation in definite logic programs is a sequence of propositional definite logic programs, plus an instantiated query. Following Boole and using linear (not integer) programming as the propositional proof procedure, this says query evaluation in a definite predicate logic program is equivalent to asserting that one of the constructed (instantiated) sequences of linear programming problems has no solution. This is a sound and complete query evaluation procedure for predicate logic programs that is the linear programming analog of ground resolution. This paper addresses the following question: Is there a sound and complete non-ground linear programming proof procedure for predicate logic programs? To answer this we need to remind ourselves what unification does. It clusters ground instances into cases which can be handled simultaneously by resolution because they are “of a similar form” for resolution. The most general unifier is used to reduce the number of such cases as far as we can. Here we have to ask a similar question: What does it mean to cluster together cases of real linear programming problems encountered during the course of the proof because they are “of similar form”? In this paper we confine ourselves to the definite clause case, and give such a sound and complete procedure for linear programming enhanced with unification. It can be best viewed as a tree of real linear programs, the shape of the tree being determined by a unification procedure. As Colmerauer [7] has proved that unification can be replaced by a linear procedure, our result implies that should an implementor so desire, s/he can replace resolution with unification completely by linear programming using our partial instantiation strategy.

Our method for evaluating logic programs proceeds as follows—first, a (non-ground) logic program \( P \) is treated as if it were a propositional logic program \( P^* \) (i.e., an atom \( A \) occurring in \( P \) is considered to be a proposition \( p_A \)). Program \( P^* \) may then be evaluated using any known mechanism for evaluating propositional logic programs \([2, 8]\). Assignments of true/false to different propositions \( p_A \) and \( p_B \) in \( P^* \) may lead to “conflicts” when \( A \) and \( B \) are unifiable, but \( p_A \) and \( p_B \) are assigned different truth values. If there are no such conflicts, then we are done. When such conflicts are present, we will articulate a precise strategy for removing such conflicts.
We will show that our strategy of

Evaluate Propositional Program → Identify Conflicts → Partially Instantiate

yields a soundness and completeness theorem for the computation of answer substitutions [14].

This paper forms part of the Cornell-Maryland LOPS project ("Logic and Optimization for Problem Solving"). When deductive databases are used to support applications requiring quick run-time responses, then performing deduction at run-time (as Prolog does) has proved inefficient. The approach taken in [2, 3, 4, 5, 17] has been to perform as much deduction as possible at compile-time, subject to availability of memory and time. Unlike previous results where the database needed to be “grounded out” prior to compilation, in this paper, grounding will not be required. Instead, an abstract tree of deductive databases will be defined. The intention is that at compile-time, all that needs to be done is to compile part of this tree—typically, as much of this tree should be constructed as possible, subject to availability of space.

2. Definite logic programs without grounding

In this section, we will show how, given any definite logic program \( P \), there is a tree \( \text{PIT}(P) \) associated with \( P \). Nodes in \( \text{PIT}(P) \) are labeled with various entities, including sets of “true” atoms (not necessarily ground). At compile-time, independently of any specific query, it is possible to construct as much of \( \text{PIT}(P) \) as is desired (or feasible depending on space availability). The soundness and completeness result we will prove establishes that \( \text{PIT}(P) \)'s construction accurately captures the set of all computed answer substitutions [14]. The construction of \( \text{PIT}(P) \) requires two concepts—the first one is that of a disagreement set which is used to generate the children of a node in \( \text{PIT}(P) \); the second is the computation of the labels associated with a given node. Both these are described in the next two subsections.

2.1. Disagreement sets

Given a set \( X \) of atoms (not necessarily ground), we use the notations \( \text{EXP}(X) \) and \( \text{GRDEXP}(X) \) to denote, respectively, the set of all instances (ground as well as non-ground) of atoms in \( X \) and the set of all ground instances of atoms in \( X \).

**Definition 1.** Two sets, \( T \) and \( F \), of atoms (not necessarily ground) are said to disagree iff there exist atoms \( A_1 \in T \) and \( A_2 \in F \) such that \( A_1 \) and \( A_2 \) are unifiable.

The disagreement set of \( T, F \), denoted \( \text{DIS}(T, F) \) is the set \( \{ \theta | \text{there exist atoms } A_1 \in T \text{ and } A_2 \in F \text{ such that } A_1 \text{ and } A_2 \text{ are unifiable via mgu } \theta \} \).
We assume that all substitutions are represented in solved form (cf. Martelli and Montanari [16, p. 261]). A set of equations (or substitutions) "is said to be in solved form if it satisfies the following conditions:

1. the equations are \( x_j = t_j, j = 1, \ldots, k \);
2. every variable which is the left member of some equation occurs only there.”

If \( T \) and \( F \) are finite sets, then it is easy to see that \( \text{DIS}(T, F) \) is finite – even if two atoms have infinitely many unifiers, there is only one mgu upto equivalence [13, 14].

Thus, the disagreement set is finite and hence can be listed. Furthermore, \( \text{DIS}(T, F) \) can be computed in polynomial time. To see this, suppose \( T = \{ A_1, \ldots, A_n \} \) and \( F = \{ B_1, \ldots, B_m \} \). Then, for each \( 1 \leq i \leq n \), and for each \( 1 \leq j \leq m \), check if \( A_i \) and \( B_j \) are unifiable; this involves \( (n \times m) \) unifiability checks. The Martelli–Montanari algorithm will find a unifier (if one exists) in linear time. Performing \( (n \times m) \) linear-time unifiability checks can be done in polynomial-time. Note that due to Colmerauer’s [7] characterization of unification in terms of linear programming, in principle, we may use the Colmerauer technique to compute \( \text{DIS}(T, F) \) instead of the Marteilli–Montanari technique.

Example 1. For example, suppose

\[
T = \{ p(X, Y), q(X, Y), r(X) \},
\]

\[
F = \{ q(a, Z), q(b, a), q(b, a), q(b, d), r(a), r(b) \}.
\]

The disagreement set of \( T \) and \( F \) is \( \{ \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6 \} \) where:

\[
\sigma_1 = \{ X = a, Y = Z \},
\]

\[
\sigma_2 = \{ X = b, Y = a \},
\]

\[
\sigma_3 = \{ X = b, Y = c \},
\]

\[
\sigma_4 = \{ X = b, Y = d \},
\]

\[
\sigma_5 = \{ X = a \},
\]

\[
\sigma_6 = \{ X = b \}.
\]

Definition 2. Substitutions \( \theta_1, \theta_2, \ldots, \theta_n \) are compatible iff \( (\theta_1 \cup \theta_2 \cup \cdots \cup \theta_n) \) is solvable [16, 14].

Intuitively, two substitutions are compatible if they do not cause any variable to be instantiated to two terms that are not unifiable.

\(^1\) It is well known [13] that when a set of term equations is solvable, then they have a most general solution that is unique up to equivalence. Two substitutions \( \theta_1 \) and \( \theta_2 \) are said to be equivalent iff there exist substitutions \( \sigma_1 \) and \( \sigma_2 \) such that \( \theta_1 \sigma_1 = \theta_2 \) and \( \theta_1 \sigma_2 = \theta_1 \), i.e., each is an instance of the other. In our definition of \( \text{DIS}(T, F) \), we assume that two different, but equivalent, mgu’s are not present in \( \text{DIS}(T, F) \).
2.2. Associating linear programs with non-ground definite databases

In [2], we demonstrated how, given a ground logic program, \( P \), it is possible to associate with \( P \), a linear program, \( \text{lc}(P) \). We now show how this may be done for non-ground logic programs.

**Definition 3.** Given a (possibly non-ground) clause \( C \) of the form

\[ A \leftarrow B_1 \land \cdots \land B_n \]

the first-order linear constraint associated with \( C \), denoted \( \text{folc}(C) \), is the following:

\[ V_A + \sum_{i=1}^{n} (1 - V_{B_i}) \geq 1. \]

Each of the linear programming variables \( V_{\text{Atom}} \) is a variable associated with \( \text{Atom} \) where \( \text{Atom} \) is not necessarily ground.

Thus, for example, if we consider the clause \( C \equiv \)

\[ p(X, Y) \leftarrow q(X, Y) \land r(X) \]

then \( \text{folc}(C) \) is the constraint:

\[ V_{p(X, Y)} + (1 - V_{q(X, Y)}) + (1 - V_{r(X)}) \geq 1. \]

**Definition 4.** Given a logic program \( P \), we construct the first-order linear constraints associated with \( P \), denoted \( \text{folc}(P) \), as follows:

1. for all clauses, \( C \in P \), \( \text{folc}(C) \in \text{folc}(P) \), and
2. for every atom \( A \) (possibly non-ground) occurring in \( P \), the constraint \( 0 \leq V_A \leq 1 \) is in \( \text{folc}(P) \).

**Definition 5.** Given a logic program \( P \), the linear programming problem associated with \( P \), denoted \( C(P) \), is the problem:

minimize \[ \sum_{A \in \text{Atoms}(P)} V_A \]

subject to the constraints in \( \text{folc}(P) \)

all variables being real.

Here, \( \text{Atoms}(P) \) is the set of all distinct atoms occurring in \( P \).

2.3. Construction of partial instantiation trees

The observant reader will note that the above formulation does not contain any "unification" constraints—in other words, atoms such as \( p(X, Y) \) and \( p(a, Z) \) are treated as two completely different atoms. This raises the question: what does it mean to assign 1 ("true") to \( V_{p(X, Y)} \) and 0 ("false") to \( V_{p(a, Z)} \)? Intuitively, we want the
assignment of 1 to an atom \( A \) to mean that \((\forall)A\) is a logical consequence of the program \( P \). However, the assignment of 0 to an atom \( A' \) does not mean anything significant\(^2\). When \( V_{p(X,Y)} \) is set to 1 and \( V_{p(A,Z)} \) is set to 0, this means that \((\forall X, Y)p(X,Y)\) is a logical consequence of \( P \). However, the fact that \( V_{p(A,Z)} \) is set to 0 indicates that "something" is wrong, namely although we have a propositional valuation, it cannot be extended in any way to a predicate logic valuation such as is needed to get an answer substitution. Disagreement sets play a role in how this conflict is corrected by partial instantiations described below.

The basic idea behind partial instantiation is this: given a logic program \( P \), we construct a propositional program \( P^* \) as follows: For every clause \( C \in P \) of the form

\[
A_1 \land \ldots \land A_n \land B_1 \land \ldots \land B_m,
\]

\( P^* \) contains the propositional clause \( C^* \equiv p_{A_1} \land \ldots \land p_{A_n} \land p_{B_1} \land \ldots \land p_{B_m} \). Then, for every constraint \( \forall V_{A_1} + \sum 1 - V_{B_1} \geq 1 \) in \( folc(P) \) we get a corresponding constraint \( \forall V_{p_{A_1}} + \sum 1 - V_{p_{B_1}} \geq 1 \) in \( folc(P^*) \). Thus, the \( V \)s are indexed with atoms (not necessarily ground) in \( P \), and this linear program is identical to the linear program associated with \( P^* \) by Bell et al. \cite{2}. It was proved in \cite{2, 3, 4} that solving the above optimization problem over the domain of real numbers yields the same optimal solution as when we solve it over the domain of integers. The key difference between what we do here, and what we did in \cite{2} is that instantiation is performed only on a "need to instantiate" basis. \( folc(P) \) and \( folc(P^*) \) are equivalent in the sense that for every solution \( s \) of \( folc(P) \), there is a solution \( s^* \) of \( folc(P^*) \) such that \( V_A \) is assigned 1 (resp. 0) by a solution to \( folc(P) \) iff \( V_{p_A} \) is assigned 1 (resp. 0) by solution \( s^* \) to \( folc(P^*) \). The disagreement set between \( T = \{A \mid V_{p_A} \text{ is set to 1 by the unique optimal solution to } folc(P^*)\} \) and \( F = \{B \mid V_{p_B} \text{ is set to 0 by the unique optimal solution to } folc(P^*)\} \) is computed. Disagreement sets are then used to "branch"—different branches partially instantiate different clauses so as to remove conflicts.

**Definition 6** (Partial Instantiation Tree associated with a Definite Logic Program). Given a logic program \( P \) (whose clauses are all standardized apart)\(^4\), we define the partial instantiation tree, \( PIT(P) \), associated with \( P \) as follows:

1. Each node, \( N \), in \( PIT(P) \) is labeled with a 4-tuple \((P_N, C_N, T_N, F_N)\) where:
   
   (a) \( P_N \) is a logic program (whose clauses may or may not be standardized apart).
   
   The root is at level 0.

   (b) \( C_N \) is the linear programming tableau associated with \( P_N \).

   (c) \( T_N \) is the set of atoms (not necessarily ground) that are assigned 1 by the optimal solution to the optimization problem\(^5\) represented by \( C_N \).

---

\(^2\) In particular, it does not mean \((\forall)\neg A' \) is a consequence of the program. Nor does it mean that \( P\neg(\forall)A' \).

\(^3\) As proved in \cite{2}.

\(^4\) Two clauses are said to be standardized apart if they share no common variable symbols. Logic program \( P \) is standardized apart if any two clauses in it are standardized apart. It is easy to see that the clauses in any logic program can be standardized apart without any change in the meaning of the program.

\(^5\) In lieu of linear programming, any method (e.g., \cite{8}) to find the set of \( \{A \mid p_A \text{ is a logical consequence of the propositional logic program } P_N^* \} \) can be used.
(d) \( F_N \) is the set of atoms that are assigned 0 by the optimal solution to this optimization problem.\(^6\)

Note that for any node, \( N \), once \( P_N \) is fixed, \( C_N, T_N \) and \( F_N \) are all uniquely determined. Hence, strictly speaking, for any node \( N \) it suffices to specify \( P_N \).

2. The root of \( \text{PIT}(P) \) is labeled with \((P, C, T, F)\) where \( P \) is the original program with all clauses in \( P \) being standardized apart.

3. If node \( N \) is labeled with \((P_N, C_N, T_N, F_N)\) and the disagreement set of \( T_N, F_N \) is empty, then \( N \) is a leaf node and has no children.

4. If node \( N \) is labeled with \((P_N, C_N, T_N, F_N)\) and \( N \)'s parent, denoted \( N' \), is labeled with \((P', C', T', F')\), and \( \text{EXP}(T_N) = \text{EXP}(T') \), then \( N \) is a leaf node and has no children. (As we will show in Proposition 1, it turns out that \( \text{EXP}(T_N) = \text{EXP}(T') \) iff each atom \( A \in (T_N - T') \) is subsumed by some existing atom \( A' \in T' \).)

5. Otherwise, for each substitution \( \theta \in \text{DIS}(T_N, F_N) \), node \( N \) (at level \( i \)) has a child (at level \( i + 1 \)) labeled with \((P_\theta, C_\theta, T_\theta, F_\theta)\) where \( P_\theta \) is obtained as follows. Let \( Q_\theta \) denote \( \{C \mid C \in P_N\} \cup P_N \). If the body of a clause in \( Q_\theta \) contains an atom \( A \), and \( A \) is subsumed by an atom in \( T_N \) for some node \( N' \) at level \( i \), then \( A \) is deleted from the body of this clause (we will refer to this as the pruning of this clause). This is carried out until all atoms occurring in clauses of \( Q_\theta \) are either deleted or are known to not satisfy this subsumption criterion. \( P_\theta \) is the set of clauses left after pruning clauses in \( Q_\theta \).\(^7\)

An important point to note is that when constructing \( \text{PIT}(P) \), clauses in the original program \( P \) are standardized apart. Consider the program \( P_N \) labeling node \( N \) where \( N \) is not the root of \( \text{PIT}(P) \). The above procedure does not standardize apart the clauses in \( P_N \) — so it is possible that clauses in \( P_N \) share variables. This is a crucial aspect of our procedure. The following examples show this.

**Example 2.** Consider the following logic program, \( P \):

\[
p(X, Y) \leftarrow q(X, Y) \& r(X).
q(a, Z). q(b, a). q(b, c).
q(b, d). r(a). r(b).
s(e). s(f). s(g).
\]

We use *facts* to denote the set of unit clauses above. The root of \( \text{PIT}(P) \) is labeled with \((P_0, C_0, T_0, F_0)\) where \( P_0 = P \) and \( C_0 \) is the linear programming tableau associated with the following minimization problem:

\[
\text{minimize } (V_{p(x, y)} + V_{q(x, y)} + V_{r(x)} + V_{q(a, z)} + V_{q(b, a)} + V_{q(b, c)})
+ V_{q(b, d)} + V_{r(a)} + V_{r(b)} + V_{s(e)} + V_{s(f)} + V_{s(g)}).
\]

\(^6\) If the linear programming approach is not used, then \( F_N \) is simply the set of all atoms \( B \) such that the propositional symbol \( p_B \) occurs in \((P_N)^*\) and \( B \) is not a logical consequence of \((P_N)^*\).

\(^7\) An important point to note here is that computation of \( \text{DIS}(T_N, F_N) \) can be done by linear programming as described in Colmerauer [7].
subject to
\[ \forall p(x, y) + (1 - \forall q(x, y)) + (1 - \forall r(x)) \geq 1, \]
\[ \forall q(a, z) \geq 1, \quad \forall q(b, a) \geq 1, \quad \forall q(b, c) \geq 1, \]
\[ \forall q(b, d) \geq 1, \quad \forall r(a) \geq 1, \quad \forall r(b) \geq 1, \]
\[ \forall s(e) \geq 1, \quad \forall s(f) \geq 1, \quad \forall s(g) \geq 1. \]

plus constraints saying all variables are greater than or equal to 0.

\( T_0 = \text{Facts} \) and \( F_0 = \{ \forall p(X, Y), \forall q(X, Y), \forall r(X) \} \). In order to compute the children of the root, we need to find the disagreement set of \( T_0 \) and \( F_0 \). This is the set \( \{ \sigma_1, \ldots, \sigma_6 \} \) where \( \sigma_1, \ldots, \sigma_6 \) are as specified in Example 1. Hence, the root has six children, one each corresponding to these six substitutions. The logic programs labeling these children are denoted by \( P_1, \ldots, P_6 \) respectively, where \( P_i \) is obtained by instantiating the rules of \( P \) by \( \sigma_i \) and adding these instantiated rules to \( P \). Thus, \( P_1 \) consists of \( P_0 \) together with the instantiated rule:

\[ \forall p(a, Z) \leftarrow \forall q(a, Z) \& \forall r(a). \]

which, after pruning results in

\[ \forall p(a, Z) \leftarrow . \]

(Note here that the program \( P_1 \) contains the clause \( \forall q(a, Z) \leftarrow \) and \( \forall p(a, Z) \leftarrow \) which are not standardized apart.) Likewise, \( P_2, \ldots, P_6 \) are obtained as shown in Fig. 1. The figure represents only parts of the tree, \( \text{PIT}(P) \).

In this example:

\( P_0 \) is the original program,

\[ P_1 = P_0 \cup \{ \forall p(a, Z) \} \quad \text{and} \quad T_1 = T_0 \cup \{ \forall p(a, Z) \}, \]

\[ P_2 = P_0 \cup \{ \forall p(b, a) \} \quad \text{and} \quad T_2 = T_0 \cup \{ \forall p(b, a) \}, \]

\[ P_3 = P_0 \cup \{ \forall p(b, c) \} \quad \text{and} \quad T_3 = T_0 \cup \{ \forall p(b, c) \}, \]

\[ P_4 = P_0 \cup \{ \forall p(b, d) \} \quad \text{and} \quad T_4 = T_0 \cup \{ \forall p(b, d) \}, \]

\[ P_5 = P_0 \cup \{ \forall p(a, Y) \} \quad \text{and} \quad T_5 = T_0 \cup \{ \forall p(a, Y) \}, \]

\[ P_6 = P_0 \cup \{ \forall p(b, Y) \leftarrow \forall q(b, Y) \} \quad \text{and} \quad T_6 = T_0, \]

\[ F_1 = F_2 = F_3 = F_4 = F_5 = F_0, \]

\[ F_6 = F_0 \cup \{ \forall p(b, Y), \forall q(b, Y) \}. \]

Note that only one pair, namely \( (T_6, F_6) \) generates new disagreements: \( \sigma_{37} = \{ Y = a \}, \sigma_{38} = \{ Y = c \}, \sigma_{39} = \{ Y = d \} \); all the other pairs generate the same set of disagreements as the original \( (T_0, F_0) \), so \( \sigma_7 = \sigma_1, \sigma_8 = \sigma_2, \ldots, \sigma_{36} = \sigma_6 \). Because of this, each of the first-level nodes has one leaf child—one corresponding to the same substitution as its parent. □
Example 3. Consider the following logic program, $P$:

$$ q(X_1) \leftarrow p(X_1, f(X_1)) \land r(g(X_1), a_1) $$

$$ r(X_2, a_1) \leftarrow r(a_1, a_2) $$

$$ p(X_3, f(X_3)) \leftarrow $$

$$ r(X_4, a_2) \leftarrow $$

We use $\text{Facts}$ to denote the set of unit clauses above. The root of $\text{PIT}(P)$ is labeled with $(P_0, C_0, T_0, F_0)$ where $P_0 = P$, $T_0 = \text{Facts}$ and $F_0 = \{q(X_1), p(X_1, f(X_1)), r(g(X_1), a_1), r(X_2, a_1), r(a_1, a_2)\}$, which gives us two disagreements $\sigma_1 = \{X_1 = X_3\}$ and $\sigma_2 = \{X_4 = a_1\}$. Hence, the root of $\text{PIT}(P)$ has two children $N_1$ and $N_2$ and the logic programs $P_1$ and $P_2$ labeling them are obtained by instantiating the rules of $P$ by the substitutions $\sigma_1$, $\sigma_2$, respectively, and then pruning. Thus, $Q_1$ (as described in the algorithm) consists of $P_0$ together with the instantiated rules:

$$ q(X_3) \leftarrow p(X_3, f(X_3)) \land r(g(X_3), a_1) $$

$$ r(a_1, a_2) \leftarrow $$
which, after pruning results in certain rules from both $P_0$ as well as the newly added instantiations being pruned to yield $P_1 =$

$$q(X_1) \leftarrow r(g(X_1), a_1)$$

$$q(X_3) \leftarrow r(g(X_3), a_1)$$

$$r(X_2, a_1) \leftarrow$$

$$p(X_3, f(X_3)) \leftarrow$$

$$r(X_4, a_2) \leftarrow$$

The sets labeling $N_1$ are $T_1 = T_0 \cup \{r(X_2, a_1)\}$ and $F_1 = \{q(X_1), q(X_3), r(g(X_1), a_1), r(g(X_3), a_1)\}$ from which we get the disagreements $\sigma_3 = \{X_2 = g(X_3)\}$ and $\sigma_4 = \{X_2 = g(X_1)\}$, so $N_1$ has two children. The node $N_2$ can be expanded in a similar way. The complete tree for the above program is shown in Fig. 2. 

**Example 4.** Consider the following (simple) logic program:

$$p(X_1, Y_1) \leftarrow q(X_1, Y_1)$$

$$q(a, Y_2) \leftarrow$$

$$q(X_2, b) \leftarrow$$

The root of PIT($P$) has $T_0 = \{q(a, Y_2), q(X_2, b)\}$ and $F_0 = \{p(X_1, Y_1), q(X_1, Y_1)\}$. This leads to two disagreements, $\theta_1 = \{X_1 = a, Y_1 = Y_2\}$ and $\theta_2 = \{X_1 = X_2, Y_1 = b\}$. Thus, two children are generated. The entire partial instantiation tree for this example is shown in Fig. 3.

The following (straightforward) lemma shows that the components of the label of any node $N$ in PIT($P$) is always finite. The is needed to ensure that DIS($T_N, F_N$) is always finite.

**Lemma 1.** Suppose $N$ is a node in PIT($P$) labeled with ($P_N, C_N, T_N, F_N$). Then $T_N, F_N,$ are finite sets of atoms, and $P_N$ is a finite logic program.

**Proof.** This can be proved by induction on the depth, $d$, of $N$.

**Base Case** ($d = 0$). As $P_N = P$ is a logic program, it contains only finitely many clauses. Hence, $P^\ast$ is a finite, propositional logic program. $T_N$ is the set of atoms $A$ such that $V_{p_i}$ is set to “true” by $folc(P^\ast)$ and is clearly finite as $P^\ast$ contains only finitely many propositional symbols. $F_N$ is the set of all propositional symbols in $P^\ast$ that are not in $T_N$—this too is immediately finite.

**Inductive Case** (true for $d = s$, show for $d = s + 1$). Suppose ($P^s, C^s, T^s, F^s$) is the label of the parent of node $N$. By the induction hypothesis, we may assume that $P^s$ is finite, as are $T^s$ and $F^s$. Hence, the disagreement set of $T^s$ and $F^s$ is finite. Consequently, $P_N$ is obtained by applying one substitution to clauses in $P^s$—hence,
Po is the original program, P1 is
q(X1) ← r(g(X1),a1)
q(X2) ← r(g(X2),a1)
r(X2,a1) ←.
p(X3,f(X3)) ←.
r(X4,a2) ←.

P2 is
q(X1) ← r(g(X1),a1)
r(X2,a1) ←.
p(X3,f(X3)) ←.
r(a1,a2) ←.
r(X4,a2) ←.

P3 is
q(X1) ←.
q(X3) ←.
p(X3,f(X3)) ←.
r(X2,a1) ←.
r(X4,a2) ←.
r(g(X1),a1) ←.

P4 is
q(X1) ←.
q(X3) ←.
p(X3,f(X3)) ←.
r(X2,a1) ←.
r(g(X1),a1) ←.

P5 is
q(X1) ←.
p(X3,f(X3)) ←.
r(X4,a2) ←.
r(a1,a2) ←.
r(g(X1),a1) ←.

the size of PN is at most twice the size of P'. It follows now, by the same reasoning as in the base case, that T_N and F_N are finite. This completes the proof. □

Before proceeding to prove the soundness and completeness of the partial instantiation strategy, we observe the PIT(P) must be constructed level by level...
because the program labeling a node at level \((i + 1)\) depends upon the set of atoms known to be true in all nodes at level \(i\). From an implementation viewpoint, this means that all the \(T\)-sets labeling a node can be stored, not necessarily in the node, but rather in a standard relational database system (alternatively, this can be viewed as a look-up table). However, the detailed description of indexing schemes to organize such look-up tables is beyond the scope of the current paper.

3. Soundness and completeness results

In this section, we show that the partial instantiation tree construction described above is sound and complete w.r.t. computation of answer substitutions. Recall that if \(A\) is an atom, and \(\theta\) is a substitution, then \(\theta\) is said to be a correct answer substitution for \(A\) w.r.t. program \(P\) iff \(P \models (\forall)A\theta\). The proof uses the propositional logic program \(P^*\) described earlier. We assume that the reader is familiar with the standard fixpoint operator, \(T_p\), associated with a logic program \(P\) [14].
Theorem 1 (Soundness). Suppose $P$ is a logic program, and $\text{PIT}(P)$ is the partial instantiation tree associated with $P$. If $A$ is an atom (not necessarily ground) in $T_N$ for some node $N$, then $P \vdash (\forall) A$.

Proof. The proof is by induction on the depth $d$ at which node $N$ occurs in the tree.

Base Case for Outer Induction ($d = 0$). In this case, $N$ is the root node, and hence, $A \in T$. As $V_A$ is assigned $1$ by the unique optimal solution to $\text{folc}(P)$, it follows that $V_{\rho_A}$ is assigned $1$ by the unique optimal solution to $\text{folc}(P^*)$. Hence, by the soundness theorem for computing propositional logic programs by linear programming proved in [2], it follows that there exists an integer $k$ such that $p_A \in T_{p^*} \uparrow k$. We proceed by induction on $k$:

Base Case for Inner Induction ($k = 1$). Then $p_A \in T_{p^*} \uparrow 1$ which means that $p_A \leftarrow$ is a unit clause in $P^*$. But then $A \leftarrow$ is a unit clause in $P$. Hence, $P \vdash (\forall) A$.

Inductive Case for Inner Induction ($k = s + 1$). $p_A \in T_{p^*} \uparrow (s + 1)$ means that there is a propositional (as $P^*$ is propositional) clause $C^*$ in $P^*$ of the form

$$p_A \leftarrow p_{B_1} \& \cdots \& p_{B_n}$$

such that $\{p_{B_1}, \ldots, p_{B_n}\} \subseteq T_{p^*} \uparrow s$. By the (inner) induction hypothesis, $P \vdash (\forall) B_i$ for all $1 \leq i \leq n$. Furthermore, $P$ contains a clause $C$ of the form

$$A \leftarrow B_1 \& \cdots \& B_n$$

corresponding to $C^*$. It follows immediately that $P \vdash (\forall) A$.

This completes the proof of the base case for the outer induction.

Inductive Case for Outer Induction ($d = d' + 1$). Let $N_0, N_1, \ldots, N_{d-1}, N_d = N$ be the nodes on the path (of length $d$) from the root of $\text{PIT}(P)$ to node $N$. The subtree rooted at $N_1$ is identical to $\text{PIT}(P_{N_1})$—this subtree represents the partial instantiation tree associated with the logic program $P_{N_1}$ labeling an immediate child of the root of $\text{PIT}(P)$. By the (outer) induction hypothesis, and as node $N$ is at depth $(d - 1)$ in the tree $\text{PIT}(P_{N_1})$, it follows that $P_{N_1} \vdash (\forall) A$. But $P_{N_1}$ is obtained from $P_{N_0} = P$ by adding some instances of clauses already in $P$, and deleting from the body of these clauses, atoms known to be true. Consequently $P_{N_0}$ and $P_{N_1}$ are logically equivalent and hence, $P_{N_0} \vdash (\forall) A$ i.e., $P \vdash (\forall) A$. □

Suppose $(A_1 \& \cdots \& A_n)$ is a conjunction of atoms such that $P \vdash (\forall)(A_1 \& \cdots \& A_n)$. The following completeness result asserts that in that case there is a level$^8$ in the tree $\text{PIT}(P)$ such that for all $1 \leq i \leq n$, $A_i \in \text{EXP}(T_{N_i})$ for some node $N_i$ occurring at that level in the tree, i.e., $T_{N_i}$ contains an atom that is more general than $A_i$.

---

$^8$ We use the convention that the root is at level 0. If node $N$'s parent node is at level $i$, then node $N$ is said to be at level $(i + 1)$. 

---
Theorem 2 (Completeness). Suppose \((A_1 \& \cdots \& A_n)\) is a conjunction of atoms such that \(P \models (\forall)(A_1 \& \cdots \& A_n)\). Then there is a level \(\ell\) in \(\text{PIT}(P)\) such that
\[
\{A_1, \ldots, A_n\} \subseteq \bigcup_{\text{level}(N_1)=\ell} \text{EXP}(\mathcal{T}_{N_1})
\]

Proof. As \(P \models (\forall)(A_1 \& \cdots \& A_n)\), there exists an integer \(k > 0\) such that \(\{A_1, \ldots, A_n\} \subseteq \text{EXP}(W_p \uparrow k)\) where \(W_p\) is the non-ground fixpoint operator defined by Falaschi et al. \([9, pp. 998-999]\)^9.

\(W_p \uparrow k \equiv (\forall)(A_1 \& \cdots \& A_n)\) implies that for each \(1 \leq i \leq k\), there exists a clause \(C_i \equiv A_i \leftarrow B_{1i}^i \& \cdots \& B_{mi}^i\)
in \(P\) having an instance (not necessarily ground), \(C_i \theta_i\) such that
(a) \(A_i\) is an instance of \(A_i\); and
(b) \(W_p \uparrow (k - 1) \equiv (\forall)(B_{1i}^i \& \cdots \& B_{mi}^i)\theta_i\).

We show by induction on \(k\), that there exists a level \(\ell\) such that
\[
\{A_1, \ldots, A_n\} \subseteq \bigcup_{\text{level}(N_1)=\ell} \text{EXP}(\mathcal{T}_{N_1})
\]

Base Case \((k = 1)\). Then for each \(1 \leq i \leq n\) there exists a clause \(C_i \equiv A_i \leftarrow P_{Ai}\) in \(P\) such that \(A_i\) is an instance of \(A_i\). Thus, there is a propositional unit clause \(C^*\) in \(P^*\) of the form \(p_{A_i}\). By the completeness of the linear programming computation of propositional define databases \([2]\), it follows that \(V_{p_{A_i}}\) is set to 1 by \(\text{falc}(P^*)\). Therefore, the variable \(V_{A_i}\) corresponding to \(A_i\) is set to 1 by the optimization problem \(CN\) where node \(N\) is the root of \(\text{PIT}(P)\). Hence, all instances of \(A_i\) are in \(\text{EXP}(T_N)\). This completes the proof of the base case as the 0th level is the level satisfying the desired property.

Inductive Case \((\text{true for } k = s, \text{ prove for } s + 1)\). For each \(1 \leq i \leq n\), there exists a clause \(C_i\) in \(P\) of the form \(A_i \leftarrow B_{1i}^i \& \cdots \& B_{mi}^i\) having an instance (not necessarily ground) of the form \((A_i \leftarrow B_{1i}^i \& \cdots \& B_{mi}^i)\theta_i\) such that
1. \((\forall)\text{Conj}\) is true in \(\text{EXP}(W_p \uparrow k)\) where \(\text{Conj} = \bigwedge_{i=1}^{n} ((B_{1i}^i \& \cdots \& B_{mi}^i)\theta_i)\), and
2. \(A_i\) is an instance of \(A_i\).

By the induction hypothesis, there is a level \(\ell_i\) in \(\text{PIT}(P)\) such that for every \(B_{ji}^i\theta_i\) occurring in \(\text{Conj}\), \(B_{ji}^i\theta_i \in \bigcup_{\text{level}(N_1)=\ell_i} \text{EXP}(\mathcal{T}_{N_1})\) where \(\gamma_{ji}^i\) is a more general substitution than \(\theta_i\), i.e., for all \(i, j, \theta_i = \gamma_{ji}^i \sigma_{ji}^i\) for some substitution \(\sigma_{ji}^i\) (Fig. 4 shows these substitutions.) Furthermore, for any fixed \(i, \gamma_{1i}^i, \ldots, \gamma_{mi}^i\) are compatible (as \(\theta_i\) is a common instance of each of them). Let \(\lambda_i\) be the most general solution of \(\gamma_{1i}^i \cup \cdots \cup \gamma_{1i}^i\); hence, \(\theta_i\) is an instance of \(\lambda_i\). It follows, by the pruning process described in Step 5 of the construction of \(\text{PIT}(P)\), that for any fixed \(i, 1 \leq i \leq n\), the unit clause
\[
(A_i) \lambda_i \leftarrow
\]

---

9 Given a set \(X\) of atoms, \(W_p(X) = \{A' \mid A' \leftarrow B_{1i}^i \& \cdots \& B_{ji}^i\in X\text{ such that }v\text{ is the mgu of }\gamma_{ji}^i\text{ and }\gamma_{ji}^i\text{ and }A' = Av\}\).

10 Note however that for two atoms \(B_{ji}^i\) and \(B_{ji}^j\) in the body of clause \(C_i\), it may very well be the case that \(\gamma_{ji}^i\) and \(\gamma_{ji}^j\) may be distinct substitutions—however, \(\theta\) is a common instance of both \(\gamma_{ji}^i\) and \(\gamma_{ji}^j\).
Fig. 4. Substitutions in the Completeness Proof.

is a unit clause in the program labeling a child of a node at level \( \ell = (\ell_0 + 1) \). But, as \( A_i \) is an instance of \( A'_i \theta_i \), and as \( (A'_i) \lambda_i \) is more general than \( A'_i \theta_i \), it follows that \( A_i \) is an instance of \( (A'_i) \lambda_i \); it follows immediately that \( A_i \) is in \( \text{EXP}(T_{CH_i}) \) for some node \( CH_i \) at level \( \ell = (\ell_0 + 1) \). This is true for each \( 1 \leq i \leq n \). This completes the proof. \( \Box \)

**Important Note.** The only place in the Soundness and Completeness proofs where the linear programming method of [Z] has been used is (for a given node \( N \)) is finding the set of all atoms \( A \) such that \( p_A \) is a logical consequence of \( (P,)^* \) where \( P_N \) is the logic program labeling node \( N \). Consequently, both the soundness and completeness theorem continue to hold if we replace the linear programming component of our tree construction by any other algorithm that finds the set of atoms \( A \) such that \( p_A \) is a logical consequence of the propositional logic program \( (P_N)^* \).

The monotonicity lemma below shows that as we go “deeper and deeper” down any path in the partial instantiation tree, the sets \( T \) labeling the nodes get larger and larger.

**Lemma 2 (Monotonicity Lemma).** Suppose \( N_0, N_1, N_2, \ldots, N_i, N_{i+1}, \ldots \) is a path in the partial instantiation tree, \( \text{PIT}(P) \), associated with logic program \( P \). Let node \( N_i \) be labeled with \( (P_i, C_i, T_i, F_i) \). Then for all \( i \geq 0, T_i \subseteq T_{i+1} \). Hence, \( \text{EXP}(T_0) \subseteq \text{EXP}(T_1) \subseteq \text{EXP}(T_2) \subseteq \cdots \).

**Proof.** Note that by the pruning construction, and the partial instantiation strategy, for each clause in \( P_i^* \), there is a clause in \( P_{i+1}^* \) which subsumes it. Hence, \( T_{P_i} \uparrow \omega \subseteq T_{P_{i+1}^*} \uparrow \omega \). Recall [14] that \( T_{P_i} \uparrow \omega \) (resp. \( T_{P_{i+1}^*} \uparrow \omega \)) is the set of propositions that are logical consequences of \( P_i^* \) (resp. \( P_{i+1}^* \)). By the Completeness of the propositional linear programming computation procedure in [2], it follows that

\[
T_i = \{ A | p_A \in T_{P_i} \uparrow \omega \} \\
\subseteq \{ A | p_A \in T_{P_{i+1}^*} \uparrow \omega \} = T_{i+1}.
\]

This completes the proof. \( \Box \)
Clause 4 in the definition of partial instantiation trees necessitates that we be able to check whether \( \text{EXP}(T_N) = \text{EXP}(T') \) where \( N \) is some node in the tree, and \( N' \) is its parent. The Monotonicity Lemma can be used to devise a simple algorithm to perform this check.

**Proposition 1.** Suppose \( N \) is a node in \( \text{PIT}(P) \) labeled with \((P_N, C_N, T_N, F_N)\) and \( N' \), its parent node, is labeled with \((P', C', T', F')\). Then, \( \text{EXP}(T_N) = \text{EXP}(T') \) iff for each atom \( A \in (T_N - T') \) there is an atom \( B \in T' \) such that \( A \) is subsumed by \( B \).

**Proof.** (\( \Rightarrow \)) Suppose \( \text{EXP}(T_N) = \text{EXP}(T') \) and \( A \in (T_N - T') \). Hence, \( A \in T_N \subseteq \text{EXP}(T_N) = \text{EXP}(T') \). Therefore, \( A \in \text{EXP}(T') \). By definition of \( \text{EXP} \), there must be an atom \( B \in T' \) such an atom \( B \) subsumes \( A \), i.e., \( A = B\theta \) for some substitution \( \theta \). But then \( A \) is subsumed by \( B \).

(\( \Leftarrow \)) Suppose that for each atom \( A \in (T_N - T') \) there is an atom \( B \in T' \) such that \( A \) is subsumed by \( B \). We need to show that \( \text{EXP}(T_N) = \text{EXP}(T') \). By the Monotonicity Lemma, and as \( N' \) is the parent of \( N \), we know immediately that \( T' \subseteq T_N \). Hence, it follows immediately that \( \text{EXP}(T') \subseteq \text{EXP}(T_N) \).

To show that \( \text{EXP}(T_N) \subseteq \text{EXP}(T') \), suppose \( A \in \text{EXP}(T_N) \). Then there is an atom \( B \in T' \) such that \( B \) subsumes \( A \), i.e., \( A = B\theta \) for some substitution \( \theta \). Consequently, \( A \in \text{EXP}(T') \). \( \square \)

The above proposition allows us to conclude that checking whether \( \text{EXP}(T_N) = \text{EXP}(T') \) can be done in polynomial-time (w.r.t. the size of \( T_N \) and \( T' \)). Suppose \( T^* = \{B_1, \ldots, B_m\} \) and \( (T_N - T^*) = \{A_1, \ldots, A_s\} \). For each \( A_i, 1 \leq i \leq m \), check if \( A_i \) is an instance of any atom in \( T^* \). This takes \( m \) checks. As there are \( n \) atoms, to determine, for each atom in \( (T_N - T^*) \), whether there is an atom in \( T^* \) subsuming it takes a total of \( (n \times m) \) "instance" checks. It is straightforward to verify that instance-checking can be done in linear-time. Hence, the check in Step 4 of the construction of \( \text{PIT}(P) \) can be achieved in polynomial-time.

Finally, a reader may wonder: "Can the tree \( \text{PIT}(P) \) be infinite?" Observe that \( \text{PIT}(P) \) is finitely-branching because, each node \( N \) in \( \text{PIT}(P) \) is labeled with a finite propositional logic program, \( P^*_N \). Thus, the disagreement set, \( \text{DIS}(T_N, F_N) \) is finite as well. As each disagreement set generates one, and only one child of \( N \), it follows that \( N \) is finitely branching. It is easy to see by example that the tree \( \text{PIT}(P) \) can contain infinite branches. Deduction from Horn clauses can enumerate all recursively enumerable sets. If \( \text{PIT}(P) \) always had only finite branches, then determining membership in any r.e. set would be decidable. However, it turns out that for definite Datalog programs, \( \text{PIT}(P) \) is always finite.

**Theorem 3 (Finiteness Theorem for Datalog Programs).** Suppose \( P \) is a function-free definite logic program. Then: \( \text{PIT}(P) \) is a finite tree. (Hence, construction of \( \text{PIT}(P) \) always terminates for function-free definite logic programs, \( P \)).

\(^{11}\) Recall that when multiple equivalent mgu's exist, only one is retained in \( \text{DIS}(T_N, F_N) \) -- redundant mgu's are discarded.
Proof. By Konig's Lemma, and as PIT(P) is finitely-branching, it suffices to show that PIT(P) contains no infinite branches. We will prove our result by contradiction. Suppose \( \mathcal{B} = N_0, N_1, N_2, \ldots \) is an infinite branch in PIT(P). Then, by condition (4) of the definition of PIT(P), it follows that:

\[
\text{EXP}(T_0) \subset \text{EXP}(T_1) \subset \text{EXP}(T_2) \subset \cdots,
\]

i.e., the sets \( \text{EXP}(T_i) \) form a strictly increasing sequence (where \( T_i \) is the \( T \)-set labeling the node \( N_i \)).

Let \( P_0 = P \) be the initial logic program labeling the root, \( N_0 \), of PIT(P). \( P \) has all its clauses standardized apart, and contains only finitely many atoms (ground and non-ground) occurring in it (i.e., the number of propositional symbols in \( P^* \) is finite). Let \( At(P_0) \) be the set of atoms occurring in \( P_0 \). Observe that at later stages in the construction of PIT(P) clauses may have the same variables occurring in them (i.e., they are not forcibly standardized apart). Let \( P_i \) be the logic program labeling \( N_i, i \geq 0 \). Each atom \( A \) in \( At(P_i) \) may be instantiated only in a finite number of ways—by replacing any variable in \( A \) by another variable occurring in \( P_0 \), or by replacing one or more variables in \( A \) by a constant in \( P_0 \) (as function symbols do not occur in \( P \)). Thus, the total number of instantiations of atoms in \( At(P_0) \) is finite. As, for each \( i \geq 0 \), \( \text{EXP}(T_i) \) is a subset of the set of such instantiations of atoms in \( At(P_i) \), it follows that there must be an integer \( j \) such that \( \text{EXP}(T_j) = \text{EXP}(T_{j+1}) \), thus contradicting the assertion that

\[
\text{EXP}(T_0) \subset \text{EXP}(T_1) \subset \text{EXP}(T_2) \subset \cdots
\]

is a strictly increasing sequence.

This completes the proof. \( \square \)

Example 5. To see a very simple example of how construction of PIT(P) terminates even in the case of function-free logic programs containing cycles, consider the logic program:

\[
\begin{align*}
p(a, b) & \leftarrow \\
p(X, Y) & \leftarrow p(Y, X)
\end{align*}
\]

This program is cyclical when its ground instantiation is considered (e.g. when \( X \) and \( Y \) are instantiated to the same constant). PIT(P) is shown in Fig. 5. \( \square \)

As a final note, we observe that PIT(P) may have two nodes in the tree that are labeled with the same program, i.e., it is possible that we have distinct nodes \( N_1 \) and \( N_2 \) such that \( P_{N_1} = P_{N_2} \). For example, given compatible substitutions \( \theta_1, \theta_2 \) in the disagreement set at a node \( N \), we may end up with two paths—one by instantiating \( P_N \) by \( \theta_1 \) and then by \( \theta_2 \); the second by instantiating \( P_N \) first by \( \theta_2 \) and then by \( \theta_1 \). The nodes \( N_1, N_2 \) at the ends of these two paths may well be labeled by the same instantiated program. It is clear that pruning the subtree rooted at one of these two
nodes will eliminate the construction of a redundant subtree. Avoiding duplicate expansion of nodes labeled with the same instantiated program can be easily achieved by constructing the tree, by using a standard \( A^* \) tree construction/search algorithm and eliminating duplicate nodes from the “open” list maintained by \( A^* \). Discussion of \( A^* \) is beyond the scope of this paper—the interested reader can consult [19].

4. Discussion: some extensions of this framework

In the preceding sections, we have given a detailed account of how our tree-based construction procedure can be used as a uniform framework for definite-clause logic programming. We are extending this framework in a number of ways, some of which are briefly outlined below. Detailed descriptions of these extensions will be reported in forthcoming papers by the authors.

4.1. First-order theorem proving

Given a set \( S \) of clauses (standardized apart), we can determine the satisfiability of \( S \) by constructing a tree of sets of propositional clauses as follows: the root of the tree is labeled with \( S^* \) (obtained by replacing, in \( S \), all atoms of the form \( A \) by the
proposition \( p_A \), a set of constraints corresponding to \( S^* \), together with the variables assigned “true” (resp. false) by the solution (if one exists) optimizing the objective function \( \min \sum_{A \text{ an atom}} V_{p_A} \). There are three cases to consider: (1) If no solution exists, then \( S \) is unsatisfiable. (2) If a solution exists, and there are no disagreements between the “true” atoms and the “false” atoms, then \( S \) is satisfiable. (3) Otherwise (i.e., a solution exists, but there are disagreements), we need to branch in a manner analogous to the definite logic programming case.

4.2. Minimal and stable model computations

Given a logic program \( P \) containing disjunctive heads, [2] shows how the minimal models of \( P \) may be computed. The computation procedure for definite logic programs outlined here can be extended to disjunctive logic programs as follows: the main difference is that at each node \( N \) in the partial instantiation tree, we associate a set of triples—each triple is of the form \((P_{N_i}, C_{N_i}, S_{N_i})\) where \( P_{N_i} \) is a disjunctive logic program, \( C_{N_i} \) is the integer linear programming tableau associated with \( P_{N_i} \), and \( S_{N_i} \) is a set of pairs of the form \((T_{N_j}, F_{N_j})\) where:

1. For each \( T_{N_j} \), there is a minimal model, \( M_j \), of \( P_{N_j} \), such that \( T_{N_j} = \{ A \mid p_A \in M_j \} \).
   (Note that the minimal models of \( P_{N_i} \) can be computed using \( C_{N_i} \) and the integer linear programming algorithm described by Bell et al. [2].)
2. \( F_{N_j} = \{ A \mid A \text{ is an atom occurring in } P_{N_i} \text{ such that } A \notin T_{N_j} \} \).

Branching is somewhat more complex than in the definite logic programming case, and we will not go into details here. Once a minimal model, \( M \), of \( P \) has been generated, we can check if it is stable by performing a non-ground version of the stable model transform that we have defined [15], computing, using the method described in this paper, the least Herbrand model, \( M' \), of the transformed non-ground program, and checking if \( M \) and \( M' \) coincide. This generalizes the propositional linear programming computation of stable and minimal models described in [5, 3, 4].

5. Conclusions

Prolog’s strategy of performing all deductions at run-time and doing almost no pre-processing at compile-time is very inappropriate for many applications where fast run-time responses are critical. This desideratum, when coupled with the need for easy expression of domain knowledge, suggests that we should perform deduction at compile-time and store a model (or models, where appropriate) in a suitably indexed

---

\( ^{12} \) The constraint corresponding to the propositional clause \((A_1 \lor \cdots \lor A_n \lor \neg B_1 \lor \cdots \lor \neg B_m)\) is \((\sum_{i=1}^{n} V_{A_i} + \sum_{j=1}^{m} (1 - V_{B_j})) \geq 1 \). In addition, there are constraints saying that all \( V_{\cdots} \) are integer variables lying between 0 and 1 (inclusive).

\( ^{13} \) In the full first-order case, when function symbols are allowed, then there may be a \( \Pi^1_1 \)-complete class of stable models and hence, branching would be infinite.
data structure (e.g. relational tables). In previous work [2, 3, 4, 5, 17], the Cornell-Maryland LOPS ("Logic and Optimization for Problem Solving") project has studied various ways of computing models and other structures that characterize the meaning of nonmonotonic deductive databases.

These methods used an extension of the mixed integer linear programming paradigm for logical deductions that was pioneered by Robert Jeroslow. One of the main problems with existing methods for performing logical deductions using OR techniques has been the infamous "grounding" problem—the translation of propositional logic clauses to optimization problems was well known, but nothing akin to Robinson’s Lifting Lemma was known that extended these translations to the first order case. Jeroslow [10] recognized this problem, but was unable to get a satisfactory lifting, to the first-order case, of the soundness and completeness results for propositional logic programs.

In this paper, we have obtained results that show how to compute the set of atoms A such that (V)A is a logical consequence of a logic program P. We do this by uniformly treating P as a propositional definite program P*, and then "branching" when the propositional logic program yields incompatible true/false assignments. We have proved that our partial instantiation strategy is sound and complete. Furthermore, it applies to all logic programs including those that contain function symbols. More importantly, our strategy is in no way tied to the linear programming paradigm—it can be used in conjunction with any method which, given a propositional logic program P*, will determine the set of all propositions that are consequences of P* (e.g. [8]). However, should a Prolog implementor so desire, s/he can completely replace both resolution and unification by linear programming in view of our partial instantiation strategy which shows how resolution can be viewed as linear programming, and Colmerauer’s results [7] which show how unification can be viewed as linear programming.

In ongoing work, we are extending our current implementation [2, 3, 4] to implement the methods described here.

Acknowledgements

We have benefited from conversations with Raymond Ng and J.B. Remmel.

References

[1] G. Boole, An Investigation into the Laws of Thought, on which are Founded the Mathematical Theories of Logic and Probability (1857).


