Handling Negation in General Deductive Databases: A Program Transformation Method

Weiling Li, Komal Khabya, Ming Fang, and Rajshekhar Sunderraman

Department of Computer Science
Georgia State University
Atlanta, Georgia 30303, U.S.A
email: {wli16,kkhabya1,mfang1}@student.gsu.edu, raj@cs.gsu.edu

Abstract
In this paper, we present a program transformation approach to compute the stable models of general deductive databases. Stable models have important applications in knowledge representation, database integration and repairs as well as in efficiently solving NP-complete problems. The method introduced in the paper first transforms the given deductive database with arbitrary negation into a “semantically” equivalent deductive database with “limited” negation. It then computes the “Fitting” model (a weaker form of the well-founded model) in a bottom-up manner. The Fitting model is a unique 3-valued model and the proposed method uses this model as a starting point to generate-and-test models for stability. Experimental analysis shows a large speed-up compared to generating the stable models from definition.

The transformation algorithm introduces two new predicates, $p_{\text{plus}}$ and $p_{\text{minus}}$, for each predicate $p$ in the original deductive database. These new predicates collect the positive and negative conclusions for the predicates from the database defined by the Fitting model. The transformed database does not contain the original negations as they are replaced by the plus and minus versions of the predicates. Although the transformation algorithm introduces negations, these are easily handled by traditional bottom up evaluators as these negations are always present in a limited manner.

1 Introduction
Deductive databases were introduced over 30 years ago ([5, 6, 7]) as a powerful extension to the relational data model with recursive views. A deductive database consists of a set of facts that correspond to a relational database and a set of logical rules that define new predicates that correspond to relational views. However, unlike relational databases where the views are restricted to be non-recursive, deductive databases can express recursive views. For example, the following is a deductive database expressing data related to a graph and a recursive view defining the predicate $\text{path}$:

\[
\begin{align*}
\text{edge}(a, b). \\
\text{edge}(b, c). \\
\text{edge}(c, d). \\
\text{path}(X, Y) :&= \text{edge}(X, Y). \\
\text{path}(X, Y) :&= \text{edge}(X, Z), \text{path}(Z, Y). \\
\end{align*}
\]

Deductive rules with arbitrary negation in their bodies are much more expressive and can represent many views that are not possible with rules without negation or with restricted negation such as stratified negation, where negation is not allowed within a recursive view. Deductive databases with arbitrary negation in the body of rules have important applications in knowledge representation, data integration and database repairs, and many other emerging areas.

The semantics of deductive databases with negation have been studied in great detail over the past several decades and two popular semantics have emerged. These are the well founded model semantics ([15]) and the stable model semantics ([8]). These semantics are well understood but there has not been sufficient research in devising efficient methods to compute these models. In [2], a bottom-up algebraic approach to compute the Fitting model (a weaker version of the well-founded model) was proposed which utilized a paraconsistent relational model and algebra ([1]). The paraconsistent data model extends the traditional relational model with explicit negation by allowing both positive and negative facts to be stored in paraconsistent relations. The relational algebraic operators were appropriately extended to operate on paraconsistent relations.

In this paper we use some of the ideas from the paraconsistent relational model and algebra to devise a program transformation method which transforms...
a deductive database with arbitrary negation into a Fitting-model equivalent deductive database in which the negation is present in a very controlled manner. The transformed database has several advantages including the possibility of using traditional bottom up evaluators that have been proven to be efficient in computing the desired models of the database.

The unique 3-valued Fitting model generated in a bottom-up manner provides the starting point for a generate-and-test procedure to compute the different stable models of the original deductive database. The “unknowns” are assigned truth values and the resulting complete model is tested for stability. Experimental results indicate a considerable speed-up compared to computing the stable models from definition.

The paper is organized as follows: Section 2 presents some background information necessary to follow the rest of the paper, Section 3 introduces the database transformation algorithm along with examples, Section 4 presents the stable model computation approach, and Section 5 discusses the experimental results. Finally, Section 6 presents concluding remarks and possible directions for future work.

2 Background

In this section we give a brief overview of general deductive databases and two popular semantics: the Fitting model and the Stable model. For a detailed exposition the reader is referred to ([4] and [8]).

We assume an underlying language with a finite set of constant, variable, and predicate symbols, but no function symbols. A term is either a variable or a constant. An atom is of the form \(p(t_1, \ldots, t_n)\), where \(p\) is a predicate symbol and the \(t_i\)'s are terms. A literal is either a positive literal \(A\) or a negative literal \(\neg A\), where \(A\) is an atom. For any literal \(l\) we let \(l'\) denote its complementary literal, i.e. if \(l\) is positive then \(l' = \neg l\), otherwise \(l = \neg l'\).

**Definition 1:** A general deductive database is a finite set of clauses of the form

\[
a \leftarrow l_1, l_2, \ldots, l_m
\]

where \(a\) is an atom, \(m \geq 0\) and each \(l_i\) is a literal.

A term, atom, literal, or clause is called ground if it contains no variables. A ground instance of a term, atom, literal, or clause \(Q\) is the term, atom, literal, or clause, respectively, obtained by replacing each variable in \(Q\) by a constant. For any general deductive database \(DB\), we let \(DB^*\) denote the set of all ground instances of clauses in \(DB\). Note that since the underlying language has no function symbols, unlike logic programs, \(DB^*\) is always finite.

The Herbrand Base of the underlying language is the set of all ground atoms. Any subset of the Herbrand Base is termed a Herbrand interpretation (atoms in the interpretation are assumed to be true and those outside the interpretation are assumed to be false). A Herbrand interpretation is a model of the database if all the facts and rules evaluate to true in the interpretation. A model is a minimal model if none of its proper subsets is a model.

**Definition 2:** A partial interpretation is a pair \(I = (I^+, I^-)\), where \(I^+\) and \(I^-\) are any subsets of the Herbrand Base.

2.1 Fitting Model

The Fitting model of a general deductive database \(DB\) is the least fixpoint of the immediate consequence function \(T^F_DB\) on consistent partial interpretations defined as follows:

**Definition 3:** Let \(I\) be a partial interpretation. Then \(T^F_DB(I)\) is a partial interpretation, given by

\[
T^F_DB(I)^+ = \{ a | \text{for some clause } a \leftarrow l_1, \ldots, l_m \text{ in } DB^*, \text{ for each } i, 1 \leq i \leq m, \\
\text{if } l_i \text{ is positive then } l_i \in I^+, \text{ and } \\
\text{if } l_i \text{ is negative then } l_i' \in I^- \}
\]

\[
T^F_DB(I)^- = \{ a | \text{for each clause } a \leftarrow l_1, \ldots, l_m \text{ in } DB^*, \text{ there is some } i, 1 \leq i \leq m, \\
\text{such that } \\
\text{if } l_i \text{ is positive then } l_i \in I^-, \text{ and } \\
\text{if } l_i \text{ is negative then } l_i' \in I^+ \}
\]

It is obvious that \(T^F_DB\) is monotonic and thus possesses a least fixpoint, which is referred to as the Fitting model for \(DB\). In other words, starting from an empty partial interpretation as the initial value of \(I\), if we repeatedly apply the \(T^F_DB\) operator to the previous value of \(I\) to generate the next value of \(I\), we are assured to reach a steady state where no more values are added to the input \(I\) in the output.

**Example 1:** Let \(DB\) be the following general deductive database:

\[
\begin{align*}
\text{r(a,c)}.
\text{r(b,b)}.
\text{s(a,a)}.
\text{p(X) :- r(X,Y), not p(Y).}
\text{p(Y) :- s(Y,a).}
\end{align*}
\]

We start with the empty partial interpretation: \(\langle \emptyset, \emptyset \rangle\).

Then, \(T^F_DB(\langle \emptyset, \emptyset \rangle)^+\) = \{ \text{r(a,c), r(b,b), s(a,a)} \}

(\(T^F_DB < \emptyset, \emptyset \rangle^-\) = \{ \text{r(a,a), r(a,b), r(b,a), r(b,b), r(c,a), r(c,b), r(c,c), s(a,a), s(a,c), s(b,a), s(b,b), s(b,c), s(c,a), s(c,b), s(c,c)} \)
\[ T_{DB}^{P}(T_{DB}^{P}(<\emptyset,\emptyset>))^+ = I \cup T_{DB}^{P}(<\emptyset,\emptyset>) \]

where \( I \) is the partial interpretation \( \langle \{p(a)\}, \{p(c)\}\rangle \). At this point we reach a steady state. Note that in the Fitting model the atom \( p(a) \) is true and the atom \( p(c) \) is false. No truth value is assigned to the atom \( p(b) \).

## 2.2 Stable Model

The stable model semantics was originally introduced by Gelfond and Lifschitz ([8]) as a two-valued model for general deductive databases. They defined a transformation, called the GL-transform, that converted a (ground) general deductive database into a negation-free (ground) deductive database based on a given Herbrand interpretation. The given interpretation was called stable if it coincided with the minimal model of the transformed database. The GL-transform is discussed now.

For any set \( S \) of atoms from the Herbrand base of a general deductive database \( DB \), let \( DB^S \) be the program obtained from \( DB \) by deleting:

1. each rule with a negative literal \( \neg B_i \) in body with \( B_i \in S \), and
2. all negative literals from bodies of remaining rules.

If \( S \) is a minimal model of \( DB^S \), then \( S \) is a stable model of \( DB \).

Consider the program

\[
p(1,2).
p(x) :- p(x,y), \neg q(y).
\]

The set of constants is \( \{1,2\} \) and the set of ground atoms is \( \{q(1), q(2), p(1,1), p(1,2), p(2,1), p(2,2)\} \). Let \( \delta \) be obtained from the program with the second rule replaced by its ground instances:

\[
p(1,2).
p(1) :- p(1,1), \neg q(1).
p(1) :- p(1,2), \neg q(2).
p(2) :- p(2,1), \neg q(1).
p(2) :- p(2,2), \neg q(2).
\]

Let \( S = \{q(2)\} \). Then \( \delta^S \) is

\[
p(1,2).
p(1) :- p(1,1).
p(2) :- p(2,1).
\]

The minimal Herbrand model of this program is \( \{p(1,2)\} \), which is different from \( S \), thus \( S \) is not stable. Now let \( S = \{p(1,2), q(1)\} \). In this case, \( \delta^S \) is

\[
p(1,2).
p(1) :- p(1,2).
p(2) :- p(2,2).
\]

The minimal Herbrand model of this program is \( \{p(1,2), q(1)\} \), i.e., \( S \). Hence \( \{p(1,2), q(1)\} \) is stable.

## 3 Database Transformation

Consider a general deductive database, \( DB \), consisting of an extensional part \( EDB \) and an intensional part \( IDB \). Without loss of generality, we will assume that there are no predicates common to \( EDB \) and \( IDB \).

For each predicate \( p \) of \( DB \), we introduce two predicates \( p_{plus} \) and \( p_{minus} \) in the transformed general deductive database \( tr(DB) \). \( p_{plus} \) will be used to collect all positive consequences of the database under predicate \( p \). Similarly, \( p_{minus} \) will collect all negative consequences of \( p \).

We will illustrate the steps of the transformation algorithm on the following general deductive database (found in [14]):

%%% Extensional Database
\[
t0(1).
g(1,2,3).
g(2,5,4).
g(2,4,5).
g(5,3,6).
\]

%%% Intensional Database
\[
t(Z) :- t0(Z). \ % rule 1
\]
\[
t(Z) :- g(X,Y,Z), t(X). \ % rule 2
\]
\[
t(Z) :- g(X,Y,Z), \neg t(Y). \ % rule 3
\]

This database instance with rules is inspired from an electronic circuit shown in Figure 1. In the circuit, each logic gate consists of one positive input and one negative input. The EDB predicate \( g(X,Y,Z) \) states the input-output relationship for a logic gate; here \( X \) is the positive input, \( Y \) is the negative input, and \( Z \) is the output. There is a second EDB predicate \( t_0 \) that is true of those input terminals that are set externally to 1. Input terminals that are set to 0 do not appear in \( t_0 \). The three rules define a predicate \( t(Z) \) which gives the value at a particular terminal. If a gate has either positive input \( X \) or negative input \( Y \), then its output is 1 or “true” if and only if either \( X \) is 1 (“true”) or \( Y \) is 0 (“false”).

![Figure 1: Circuit for Example 1 (source: [14])](image)
3.1 Transformation Algorithm

The transformation of DB is done in four steps:

Step 1: Domain Predicate:

Introduce a unique unary predicate dom. For each constant symbol, a, present in DB, output the fact:

\[ \text{dom}(a) \].

For the example database, the following facts are produced in the output of Step 1:

\[ \text{dom}(1), \text{dom}(2), \text{dom}(3), \text{dom}(4), \text{dom}(5), \text{dom}(6). \]

Step 2: Extensional Database:

For each fact \( p(a_1, \ldots, a_n) \) in EDB, output the fact:

\[ p\_plus(a_1, \ldots, a_n). \]

For each predicate \( p \) with arity \( k \) in EDB, output the rule:

\[ p\_minus(X_1, \ldots, X_k) :- \text{dom}(X_1), \ldots, \text{dom}(X_k), \text{not } p\_plus(X_1, \ldots, X_k). \]

For the example database, the following facts and rules are produced in the output of Step 2:

\[ t0\_plus(1). \]
\[ t0\_minus(X) :- \text{dom}(X), \text{not } t0\_plus(X). \]
\[ g\_plus(1,2,3). \]
\[ g\_plus(2,5,4). \]
\[ g\_plus(2,4,5). \]
\[ g\_plus(5,3,6). \]
\[ g\_minus(X,Y,Z) :- \text{dom}(X), \text{dom}(Y), \text{dom}(Z), \text{not } g\_plus(X,Y,Z). \]

Step 3: Intensional Database:

Consider a rule in IDB of the form:

\[ p(W_1, \ldots, W_l) :- q_1(X_1), \ldots, q_n(X_n), \text{not } r_1(Y_1), \ldots, \text{not } r_m(Y_m). \]

where \( p, q_1, \ldots, q_n, r_1, \ldots, r_m \) are positive predicates, \( W_1, \ldots, W_l \) are distinct variables, and \( X_1, \ldots, X_n, Y_1, \ldots, Y_m \) are vectors of variables/constants. For each such rule, perform Steps 3a and 3b.

Step 3a. Output “plus” rule:

Output the following rule for \( p\_plus \):

\[ p\_plus(W_1, \ldots, W_l) :- q_1\_plus(X_1), \ldots, q_n\_plus(X_n), r_1\_minus(Y_1), \ldots, r_m\_minus(Y_m). \]

For the example database, the following plus rules will be produced:

\[ t\_plus(Z) :- t0\_plus(Z). \]
\[ t\_plus(Z) :- g\_plus(X,Y,Z), t\_plus(X). \]
\[ t\_plus(Z) :- g\_plus(Y,Z), t\_minus(Y). \]

Step 3b. Output temporary “minus” rules:

Let \( V_1, \ldots, V_k \) be the distinct variables present in the body of the rule. We shall assume that the distinct variables present in the head of the rule are also present in the body of the rule (an assumption usually made in general deductive databases to ensure safety of rules). Let \( W_1, \ldots, W_l \) be the distinct variables in the head of the rule.

Step 3b-1:

For each positive subgoal in rule, \( q_i(X_i) \), output:

\[ \text{temp}_p_j(V_1, \ldots, V_k) :- \text{dom}(U_1), \ldots, \text{dom}(U_a), q_i\_minus(X_i). \]

where \( j \) is the unique rule number and \( \{U_1, \ldots, U_a\} \) is the set of variables present in \( \{V_1, \ldots, V_k\} \) but not in \( X_i \).

Step 3b-2:

For each negative subgoal in rule, \( \text{not } r_i(Y_i) \), output:

\[ \text{temp}_p_j(V_1, \ldots, V_k) :- \text{dom}(U_1), \ldots, \text{dom}(U_a), r_i\_plus(Y_i). \]

where \( j \) is the unique rule number and \( \{U_1, \ldots, U_a\} \) is the set of variables present in \( \{V_1, \ldots, V_k\} \) but not in \( Y_i \).

Step 3b-3:

Output the following two rules:

\[ \text{temp}_p_j_2(W_1, \ldots, W_l) :- \text{dom}(V_1), \ldots, \text{dom}(V_k), \text{not } \text{temp}_p_j(V_1, \ldots, V_k). \]
\[ p\_minus_j(W_1, \ldots, W_l) :- \text{dom}(W_1), \ldots, \text{dom}(W_l), \text{not } \text{temp}_p_j_2(W_1, \ldots, W_l). \]

For the example database, the following temporary minus rules are generated:

\[ t\_minus_1(Z) :- t0\_minus(Z). \]
\[ t\_minus_1(Z) :- \text{dom}(Z), \text{not } t\_minus_1(Z). \]
\[ t\_minus_2(Z) :- \text{dom}(Z), \text{not } t\_minus_2(Z). \]
% rule 2
\[\text{temp\_t\_2}(X,Y,Z) :- g\_minus(X,Y,Z).\]
\[\text{temp\_t\_2}(X,Y,Z) :- \text{dom}(Y), \text{dom}(Z), \text{t\_minus}(X).\]
\[\text{temp\_t\_2}(X,Y,Z) :- \text{dom}(X), \text{dom}(Y), \text{dom}(Z), \not \text{temp\_t\_2}(X,Y,Z).\]
\[\text{t\_minus\_2}(Z) :- \text{dom}(Z), \not \text{temp\_t\_2\_2}(Z).\]

% rule 3
\[\text{temp\_t\_3}(X,Y,Z) :- g\_minus(X,Y,Z).\]
\[\text{temp\_t\_3}(X,Y,Z) :- \text{dom}(X), \text{dom}(Z), \text{t\_plus}(Y).\]
\[\text{temp\_t\_3\_2}(Z) :- \text{dom}(X), \text{dom}(Y), \text{dom}(Z), \not \text{temp\_t\_3}(X,Y,Z).\]
\[\text{t\_minus\_3}(Z) :- \text{dom}(Z), \not \text{temp\_t\_3\_2}(Z).\]

Step 4. Output “minus” rules:

For each IDB predicate \( p \) defined in rules numbered \( i1,...,in \), output the following rule:

\[\text{p\_minus}(W1,...,Wl) :- \text{dom}(W1),...\text{dom}(Wl), \not \text{temp\_p\_minus}(W1,...,Wl).\]

For the example database, the following minus rules are generated:

\[\text{t\_minus}(Z) :- \text{dom}(Z),\]
\[\text{t\_minus\_1}(Z),\]
\[\text{t\_minus\_2}(Z),\]
\[\text{t\_minus\_3}(Z).\]

This ends the description of the transformation algorithm.

A bottom-up evaluation of the output program for the example database results in the following values for \( \text{t\_plus} \) and \( \text{t\_minus} \):

\[\text{t\_plus} = \{<1>,<3>\}\]
\[\text{t\_minus} = \{<2>\}\]

This is verified by computing the “minimal model” of the output program using bottom-up computation until a steady state is reached. It can also be easily verified that the model produced by our transformed program coincides with the Fitting model for the input program.

3.2 Discussion

The rationale behind the steps of the algorithm is discussed now. The main idea behind the transformation is motivated by the paraconsistent relational model and algebra that was used to compute the Fitting model of general deductive databases in [2]. The approach in this program transformation explicitly use predicates for both positive as well as negative consequences of the database.

Step 1 of the transformation algorithm introduces \( \text{dom} \), a unary predicate that collects all constants (elements of the Herbrand Universe).

Step 2 of the algorithm is also straightforward; it explicitly states that the facts specified in the \( EDB \) are positive facts and those that are missing are negative facts.

Step 3a of the algorithm is also reasonably straightforward. The “plus” component of the \( IDB \) predicate defined in the head of the rule is obtained by replacing positive body predicates by the corresponding “plus” predicates and the negative body predicates by the corresponding “minus” predicates. The reason is that for the positive body predicate to be true, a tuple (corresponding to the arguments of the predicate) must be present in the “plus” predicate and for the negative body predicate to be true, a tuple (corresponding to the arguments of the predicate) must be present in the “minus” predicate.

To understand Step 3b of the algorithm, we briefly present the paraconsistent model ([1]) and the relevant algebraic operators. A paraconsistent relation \( R \) is defined as a pair \((R^+, R^-)\), where \( R^+ \) and \( R^- \) are sets of tuples in the relational schema, where tuples in \( R^+ \) denote positive facts and tuples in \( R^- \) denote negative facts. Some of the relevant algebraic operations are shown below (note the dot on top of the paraconsistent operator - to distinguish it from the ordinary operator):

**Definition 4:** Let \( R \) and \( S \) be paraconsistent relations on scheme \( \Sigma \). Then,

(a) the union of \( R \) and \( S \), denoted \( R \cup S \), is a paraconsistent relation on scheme \( \Sigma \), given by
\[ (R \cup S)^+ = R^+ \cup S^+, \quad (R \cup S)^- = R^- \cap S^- \]

(b) the intersection of \( R \) and \( S \), denoted \( R \cap S \), is a paraconsistent relation on scheme \( \Sigma \), given by
\[ (R \cap S)^+ = R^+ \cap S^+, \quad (R \cap S)^- = R^- \cup S^- \]

(c) the complement of \( R \), denoted \( \hat{\neg} R \), is a paraconsistent relation on scheme \( \Sigma \), given by
\[ (\hat{\neg} R)^+ = R^-, \quad (\hat{\neg} R)^- = R^+ \]

If \( \Sigma \) and \( \Delta \) are relation schemes such that \( \Sigma \subseteq \Delta \), then for any tuple \( t \in \tau(\Sigma) \), we let \( t^\Delta \) denote the set \( \{t' \in \tau(\Delta) \mid t'(A) = t(A), \text{ for all } A \in \Sigma\} \) of all extensions of \( t \). We extend this notion for any \( T \subseteq \tau(\Sigma) \) by defining \( T^\Delta = \cup_{t \in T} t^\Delta \).

**Definition 5:** Let \( R \) and \( S \) be partial relations on schemes \( \Sigma \) and \( \Delta \), respectively. Then, the natural join...
Let $R$ be a paraconsistent relation on scheme $\Sigma$, and $\Delta \subset \Sigma$ be any scheme. Then, the projection of $R$ onto $\Delta$, denoted $\pi_\Delta (R)$, is a paraconsistent relation on $\Delta$, given by

$$\pi_\Delta (R) = \{ t \in \tau(\Delta) \mid t^\Sigma \subseteq R^- \}$$

where $\pi_\Delta$ is the usual projection over $\Delta$ of ordinary relations.

Now, we return to the discussion on Step 3b of the algorithm.

The first point to note is that the negative component of the “join” of paraconsistent relations corresponds to a union of extension of tuples from each of its operands. This is the reason behind producing a separate rule (using temp predicates) for each of the positive (Step 3b-1) and negative (Step 3b-2) predicates in the body of the rules, thereby implementing the “union”. It should also be noted that tuples are extended to the full schema of the body by adding $\text{dom}(X)$ in the body of the temp rules for each $X$ that is not present in the predicate.

Step 3b-3 of the algorithm can be justified by looking at the “projection” operator on paraconsistent relations. Usually, in the bottom-up methods to compute immediate consequences of the deductive database, the projection operator is used to remove the unnecessary variables from the body relation. In the definition of the negative component of the projection operator, we notice a forall-quantifier being used (implicitly in the $\subseteq$ expression). In essence, the definition states that if a particular value (for the projected variables) is associated with “all” values from the domain for the remaining variables of the body, the particular value is kept in the negative part of the projection. To implement the forall-quantifier using deductive rules, we employ a two-rule strategy with limited negations in each as seen in Step 3b-3. This is the only step in which negations are introduced in the transformed database, an important point to note. Otherwise the rest of the transformed program is negation-free. Furthermore, the negation introduced is a controlled one, i.e. it is present next to a universe of values obtained by the cartesian product of the domains. This type of negation is easily handled by bottom up evaluators.

Step 4 of the algorithm relies on the “union” operation of the paraconsistent algebra. To obtain the negative component of the output of the union of two paraconsistent relations, the intersection of the negative components is taken. The output rule in Step 4 indicates this intersection.

Here is another complete example of the transformation algorithm:

**Example 2:** Consider the following general deductive database:

```prolog
p(X,Y) :- r(X,Y), not q(Y). % rule 1
q(X) :- r(Y,X), not p(X,Y). % rule 2
```

The transformed database is:

```prolog
% Output of Step 1:
dom(1).
dom(2).
dom(3).
dom(4).
```

```prolog
% Output of Step 2:
r_plus(1,2).
r_plus(2,3).
r_plus(3,4).
r_minus(X1,X2) :-
   dom(X1), dom(X2), not r_plus(X1,X2).
```

```prolog
% Output of Step 3:
p_plus(X,Y) :- r_plus(X,Y), q_minus(Y).
p_plus(X) :- r_plus(Y,X), p_minus(X,Y).
```

```prolog
temp_p_1(X,Y) :- r_minus(X,Y).
temp_p_1(X,Y) :- dom(Y), q_plus(X).
temp_p_1_2(X,Y) :-
   dom(X), dom(Y), not temp_p_1(X,Y).
p_minus_1(X,Y) :-
   dom(X), dom(Y), not temp_p_1_2(X,Y).
```

```prolog
temp_q_2(X,Y) :- r_minus(Y,X).
temp_q_2(X,Y) :- p_plus(Y,X).
temp_q_2_2(X) :-
   dom(X), dom(Y), not temp_q_2(X,Y).
q_minus_2(X) :-
   dom(X), not temp_q_2_2(X).
```

```prolog
% Output of Step 4
p_minus(X,Y) :-
   dom(X), dom(Y), p_minus_1(X,Y).
q_minus(X) :-
   dom(X), q_minus_2(X).
```

A bottom-up evaluation of the output program results in the following values for $p_{\text{plus}}, p_{\text{minus}}, q_{\text{plus}},$ and $q_{\text{minus}}$:

```prolog
p_{\text{plus}} = \{\}
```
which coincides with the Fitting model for the input program.

The following theorem establishes the correctness of the algorithm:

**Theorem 1:** Let $DB$ be a general deductive database and let $tr(DB)$ be the output of the transformation algorithm. Then,

1. $tr(DB)$ has a unique and complete well-founded model.
2. $p(a_1, ..., a_n)$ belongs to the positive component of the Fitting model of $DB$ if and only if $p_{plus}(a_1, ..., a_n)$ belongs to the well-founded model of $tr(DB)$.
3. $p(a_1, ..., a_n)$ belongs to the negative component of the Fitting model of $DB$ if and only if $p_{minus}(a_1, ..., a_n)$ belongs to the well-founded model of $tr(DB)$.

All the variables in the negated predicates of the transformed database are always constrained to be elements of dom. This allows bottom-up computation of the head predicate to proceed by employing the “limited-complement” operator of the relational algebra. The net result is that the bottom-up computation reaches a steady state in a complete model, which coincides with the well-founded model of the transformed database.

4 Stable Model Computation

Recently, the stable models of general deductive databases have been shown to be useful in speeding up solutions to many NP-complete problems in graph theory ([12, 13]). Therefore, computing the stable models in an efficient manner is of importance. We propose a methodology that uses the database transformation described in the previous section to compute the stable models of a general deductive database. This approach will be shown to be substantially faster than the naive approach to computing the stable models that uses the basic definition of the Gelfond-Lifschitz transform.

4.1 Naive Approach

Here, we summarize the naive approach for computing the stable models shown in Figure 2. This method uses the Gelfond-Lifschitz ([8]) transformation $DB^S$ of $DB$ with respect to $S$, where $DB$ is the original deductive database and $S$ is a candidate Herbrand model. The deductive database is first compiled using a Datalog compiler (coded using Java JCup/JFlex) and a data structure is produced that contains the essence of the facts and rules. The main loop is a generate-and-test loop in which a candidate Herbrand model ($S$) is generated using the data structure as input. Next, the data structure and the candidate Herbrand model are used as inputs to produce a ground instance of the deductive database. Useless rules are eliminated in the process to minimize the size of the ground instance. The candidate model is then tested for stability. To perform the stability test, the minimal model of the resulting ground deductive database is computed in a bottom-up manner and tested to see if it coincides with the candidate Herbrand model.

4.2 Database Transformation Approach

We propose to use the transformation algorithm and the subsequent bottom up evaluation of the Fitting model as the preprocessing steps to computing the stable models. Figure 3 explains the modified approach. The “Generate Candidate Model” module is now preceded by the “Transformation” and the “Fitting Model”. The extra time spent in the preprocessing step that computes the Fitting model is offset by the big reduction in the number of candidate models generated.
4.2.1 Transformation Module

This module uses the database transformation algorithm mentioned in Section 3.1. We introduce unknown values via rules of the form:

\[
p_{\text{unknown}}(X_1, \ldots, X_k) : -
\quad \text{dom}(X_1), \ldots, \text{dom}(X_k),
\quad \text{not } p_{\text{positive}}(X_1, \ldots, X_k),
\quad \text{not } p_{\text{negative}}(X_1, \ldots, X_k).
\]

for each IDB predicate.

For the example 1, the following unknown rules are generated:

\[
t_{\text{unknown}}(Z) : -
\quad \text{dom}(Z),
\quad \text{not } t_{\text{positive}}(Z),
\quad \text{not } t_{\text{negative}}(Z).
\]

The output program for the example 1 database results in the following values for \( t_{\text{unknown}} \):

\[
t_{\text{unknown}} = \{<4>, <5>, <6>\}
\]

It coincides with the meaning of the circuit in Figure 1.

4.2.2 Fitting Model Module

After we have generated the deductive database by the transformation algorithm we use the Fixpoint semantics and apply the Fixpoint operator \( T_{DB} \) on the transformed deductive database. The application of \( T_{DB} \) gives us the Fitting model for the deductive database.

4.2.3 Generate Candidate Models Module

This module is essentially the same as before, except that it generates the candidate models by systematically “completing” the potentially incomplete Fitting model. The models for stability testing are generated from the unknown and positive values of the Fitting model. Because unknowns can be positive or negative, these unknowns can be systematically placed in the “plus” part of the model and the resulting complete model can be tested for stability.

5 Experiments

We present the experiments performed to test the efficiency of our architecture. We perform three experiments to compare the time taken to compute the stable models using our proposed architecture and a naive method of stable model computation. We use the IDB from Example 1 we discussed above with various EDBs as our logic program.

\%generate EDB facts of \( t \)
\%generate EDB facts of \( g \)

t(Z) := t0(Z).
t(Z) := g(X,Y,Z), t(X).
t(Z) := g(X,Y,Z), not t(Y).
Note that the facts in the EDB would be generated randomly from constant values in the experiments. In the experiments we keep vary the following parameters:

1. number of constants (#constants).
2. size of EDB (#facts = the number of t0_facts (#t0_facts) + the number of g facts (#g_facts))
3. the percentage of minus values (minus%) in the total number of t values

We use tables as well as graphs to show the results.

5.1 Experiment 1

Given the IDB rules we keep #t0_facts fixed to 2 and #g_facts fixed to 10, and vary the number of constants present in the program in increments of 1, starting from 4 and going up to 9.

<table>
<thead>
<tr>
<th>#constants</th>
<th>Our Approach</th>
<th>Naive Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>10735</td>
<td>4234</td>
</tr>
<tr>
<td>5</td>
<td>26125</td>
<td>13625</td>
</tr>
<tr>
<td>6</td>
<td>26249</td>
<td>41719</td>
</tr>
<tr>
<td>7</td>
<td>88688</td>
<td>132562</td>
</tr>
<tr>
<td>8</td>
<td>60952</td>
<td>374219</td>
</tr>
<tr>
<td>9</td>
<td>203421</td>
<td>1052047</td>
</tr>
</tbody>
</table>

The results of experiment 1, seen in Table 1 and Figure 4, show that our approach performs better than the Naive approach in case of larger number of constants. The number of stable models tested using n input values is $2^n$, where each model contains the EDB facts of the original deductive database, which are always true. In the Naive approach $2^{number\_of\_constants}$ models are tested, while computing the Fitting model in our approach drastically reduces the possible models for testing as we only consider the set of positive and unknown values. We can see a exponential growth for the Naive approach in Figure 3. Also we can see that in case of smaller data i.e. with 4 constants Naive approach performs better than our approach because of the overhead of using Fitting model as a preprocessing mechanism. But, for larger data i.e. even with 6 constants our approach performs much better. In the results of experiment 1 we found that the time taken for #constants = 7 is larger than that for #constants = 8, because the number of minus input values we delete during preprocessing steps also affects the running time. We will discuss more about this in experiment 3.

5.2 Experiment 2

Given the IDB rules we keep #constants fixed to 7 and vary #facts, in increments of 2, starting from 10 and going up to 20.

<table>
<thead>
<tr>
<th>#facts</th>
<th>Our Approach</th>
<th>Naive Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>53859</td>
<td>130125</td>
</tr>
<tr>
<td>12</td>
<td>36672</td>
<td>129282</td>
</tr>
<tr>
<td>14</td>
<td>59720</td>
<td>133562</td>
</tr>
<tr>
<td>16</td>
<td>54891</td>
<td>130406</td>
</tr>
<tr>
<td>18</td>
<td>94266</td>
<td>132390</td>
</tr>
<tr>
<td>20</td>
<td>99032</td>
<td>143843</td>
</tr>
</tbody>
</table>

The results of experiment 2, seen in Table 2 and Figure 5, show that as we increase the number of facts in our logic program the time taken to compute the stable models increases a little bit for the Naive approach. The percentage of minus values that will be discussed in experiment 3 affects the time taken more than the number of facts.

5.3 Experiment 3

Given the IDB rules we keep #constants fixed to 10, #t0_facts fixed to 1, and #g_facts fixed to 15, then
we check how the percent of minus values affects the running time.

Table 3: Results from Experiment 3

<table>
<thead>
<tr>
<th>minus%</th>
<th>Our Approach</th>
<th>Naive Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1651750</td>
<td>2905656</td>
</tr>
<tr>
<td>20</td>
<td>856078</td>
<td>2906328</td>
</tr>
<tr>
<td>30</td>
<td>474890</td>
<td>2932235</td>
</tr>
<tr>
<td>40</td>
<td>272187</td>
<td>2898391</td>
</tr>
<tr>
<td>50</td>
<td>156671</td>
<td>2878047</td>
</tr>
<tr>
<td>60</td>
<td>133188</td>
<td>2905781</td>
</tr>
</tbody>
</table>

As mentioned above, the time taken is related with the number of stable models tested, which is $2^n$ with n input values. The results of experiment 3 seen in Table 3 and Figure 6 show that in case of a larger percentage of minus value, our approach becomes much better than the naive approach.

From these three experiments discussed above, we can conclude that the time taken for preprocessing steps in our approach has a significant impact on the overall time to compute the stable models. In addition, when we can delete more minus values in the preprocessing steps using the Fitting model, the performance of our approach becomes even more significant.

6 Conclusion

In this paper, we have introduced a database transformation algorithm to eliminate arbitrary negations in general deductive databases and at the same time retain the meaning of the deductive database with respect to the Fitting model. The transformation enables us to use traditional bottom up evaluators for computing the meaning of general deductive databases and allows for query processing in the presence of arbitrary negations in rules. We have also shown that the database transformation method can be effectively used in computing the stable models of general deductive databases.

In future work, we propose to extend the algorithm to work with well-founded models instead of the weaker Fitting models. Also, the approach will be extended to disjunctive deductive databases ([9, 10]).

References


