Incremental Algorithms for Optimizing Model Computation Based on Partial Instantiation*

Raymond T. Ng† and Xiaomei Tian
Department of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z4,
Canada.

Abstract

It has been shown that mixed integer programming methods can effectively support minimal model, stable model and well-founded model semantics for ground deductive databases. Recently, a novel approach called partial instantiation has been developed which, when integrated with mixed integer programming methods, can handle non-ground logic programs. The goal of this paper is to explore how this integrated framework based on partial instantiation can be optimized. In particular, we develop an incremental algorithm that minimizes repetitive computations. We also develop several optimization techniques to further enhance the efficiency of our incremental algorithm. Experimental results indicate that our algorithm and optimization techniques can bring about very significant improvement in run-time performance.

keywords: incremental computation, mixed integer programming, logic programs

1 Introduction

Very active research in the past decade has led to the development of numerous methods for evaluating deductive databases and logic programs. Algorithms, such as magic sets and counting methods, have proven to be very successful for definite and stratified deductive databases [1, 2]. During the past few years, however, several new semantics for disjunctive programs and programs with negations, such as minimal models, stable models and well-founded models [18, 12, 22], have been proposed and widely studied. Recently, it has been shown that mixed integer programming methods can be used to provide a general and rather effective computational paradigm for those semantics [3, 4, 20].

*Research partially sponsored by NSERC Grants OGP0138055 and STR0134419.
†Person handling correspondence. Email: rng@cs.ubc.ca

However, like other methods that use linear or integer programming methods for logic deduction [10, 15], the paradigm proposed in [3, 4, 20] is in effect propositional, and can only deal with the ground versions of deductive databases, which are normally much larger in sizes than their non-ground versions. To solve this problem, [16, 17] very recently propose a novel approach, called partial instantiation, which combines unification with mixed integer programming (or with any other propositional deduction techniques), and which can directly solve a non-ground version of a program. Equally importantly, the approach can handle function symbols, thus making it a true logic programming computational paradigm. While we will discuss partial instantiation in greater details in Section 2, the general strategy is to alternate iteratively between two phases:

evaluation (of propositional program) $\rightarrow$ partial instantiation $\rightarrow$ evaluation ...

More specifically, the initial step begins with evaluating a given non-ground logic program P that may contain disjunctive heads and negations in the bodies as a propositional program using mixed integer programming. This generates a set of true propositional atoms and a set of false propositional atoms. The partial instantiation phase then begins by checking whether unification or "conflict resolution" is possible between atoms in the two sets. If A is an atom in the true set and B an atom in the false set, the most general unifier for A and B is called a *conflict-set* unifier. Then for each conflict-set unifier θ (there can be multiple), clauses in P are instantiated with θ and added to P for further evaluation. In other words, in the next iteration, the (propositional) program to be evaluated is $P \cup P\theta$. This process continues, until either no more conflict-set unifier is found, or the time taken has gone beyond a certain time limit [1].

The main focus of this paper is on how to optimize the run-time performance of the evaluation phase. In particular, as described in [3, 4, 20], the evaluation of program P comprises of two steps: a step to reduce the size of P, followed by the mixed integer programming step to find the models. Let us represent the operations symbolically as $model(sizeopt(P))$. As shown in [3, 4, 20], the operation $sizeopt$ to reduce the size of programs is highly beneficial to the subsequent operation of finding the models. Thus, as far as the partial instantiation paradigm is concerned, if $\theta_1, \ldots, \theta_n$ are all the conflict-set unifiers, an obvious strategy will be to compute $model(sizeopt(P \cup P\theta_1)), \ldots, model(sizeopt(P \cup P\theta_n))$ one by one. The major problem tackled in this paper is how to compute $sizeopt(P \cup P\theta_i)$ incrementally. That is, we try to optimize the evaluation phase by reusing $sizeopt(P)$ to compute $sizeopt(P \cup P\theta_1), \ldots,$ $sizeopt(P \cup P\theta_n)$. As will be shown in Example 2, our task is complicated by the fact that $sizeopt$ is not a monotonic operation. The principal contributions of this paper are:

- the development of an algorithm, called Incr, which will be formally proved to be incremental;
- the development of several optimizations which may further reduce the size of a program, save time in computing least models, and avoid processing conflict-set unifiers that are redundant;

[1] Partial instantiation may be infinite in the presence of function symbols.

- the implementation and experimental evidence showing that these algorithms and optimizations can lead to significant improvement in run-time efficiency; and
- the implementation of the entire framework that includes both the evaluation and partial instantiation phases.

Excellent work has been done on incremental view maintenance for relational, active and deductive databases [5, 6, 9, 11, 13, 14, 21, 23]. Most relevant to our work here are the proposals for deductive databases. [14] deals with recursive views; [11] is concerned with right-linear chains; [23] focuses on rules with negations; and last but not least, [13] handles rules with aggregations, recursions and negations. However, all these proposals are concerned with changes – insertions, deletions and/or updates – to the external database predicates or the base relations. As such, there are two main differences between the work presented here and the existing ones mentioned above. First, the algorithms we developed focus on handling rules inserted or deleted. Second, the operation under consideration here is not logic deduction, i.e. deducing heads from the bodies of rules. Rather, as will be discussed in greater details in Section 2, the operation *sizeopt* takes a set P of rules as input, and returns a subset $P' \subseteq P$ by deleting rules that will not be useful in subsequent model computations.

The outline of the paper is as follows. Section 2 reviews partial instantiation and the operation *sizeopt*. Section 3 presents an incremental algorithm Incr and proves that it is indeed incremental with respect to *sizeopt*. Section 4 develops several optimizations to further improve the performance of Incr and minimal model computation based on partial instantiation. Section 5 gives implementation details and presents experimental results showing the effectiveness of the algorithms and optimizations.

2 Preliminaries

2.1 Review: Partial Instantiation

As described in [16, 17], computing minimal models of logic programs by partial instantiation can be viewed as expanding and processing nodes of *partial instantiation trees*. Given a non-ground logic program P with disjunctive heads and negations in the bodies, the root node of the partial instantiation tree corresponding to P solves P directly as a propositional program. Consider an example presented in [16] where P is the program consisting of the following clauses:

$$\begin{aligned} p(X_1, Y_1) &\leftarrow q(X_1, Y_1) \\ q(a, Y_2) &\leftarrow \\ q(X_2, b) &\leftarrow \end{aligned}$$

In the root node, P is solved as the program $\{A \leftarrow B, C \leftarrow, D \leftarrow\}$, where A, B, C, D denote $p(X_1, Y_1), q(X_1, Y_1), q(a, Y_2)$ and $q(X_2, b)$ respectively. For this propositional program, the set

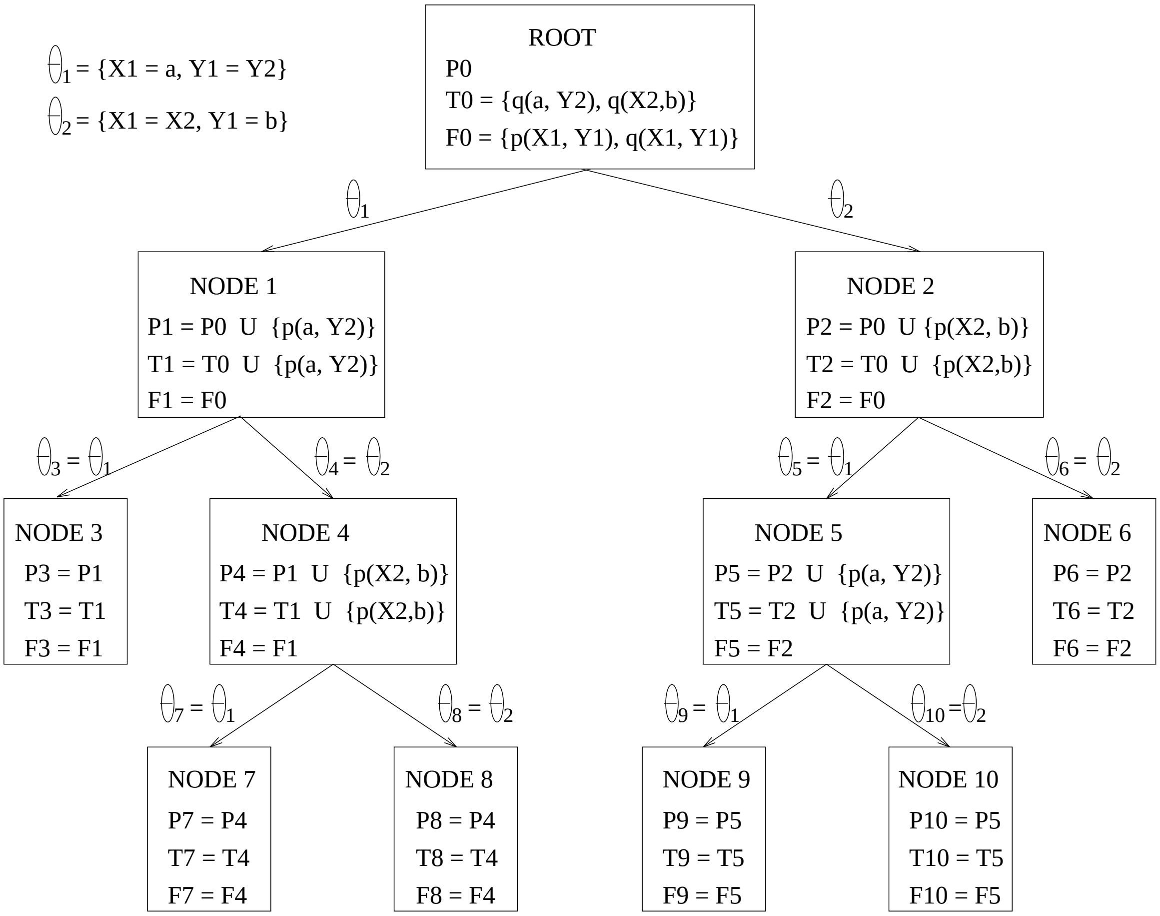

Figure 1: An Example of a Partial Instantiation Tree

of true atoms is $T = \{C, D\}$, and the set of false atoms is $F = \{A, B\}$. "Conflict resolution" then looks for unification between an atom in T with an atom in F. For our example, there are two conflict-set unifiers: a) $\theta_1 = \{X_1 = a, Y_1 = Y_2\}$, and b) $\theta_2 = \{X_1 = X_2, Y_1 = b\}$. Now for each conflict-set unifier θ_i, a child node is created which is responsible for the processing of the instantiated program $P \cup P\theta_i$. As shown in Figure 1, the root node of the tree for our example has two child nodes. One corresponds to the program $P_1 = P \cup \{p(a, Y_2) \leftarrow q(a, Y_2)\}$. The other child node corresponds to $P_2 = P \cup \{p(X_2, b) \leftarrow q(X_2, b)\}$. In the evaluation phase of P_2, P_2 again is treated as a propositional program whose true and false sets are $T_2 = \{q(a, Y_2), q(X_2, b), p(X_2, b)\}$ and $F_2 = F$. For T_2 and F_2, there are two conflict-set unifiers which are identical to θ_1, θ_2. Thus, the node for P_2 has two child nodes. Similarly, it is not difficult to verify that the node for P_1 also has two child nodes. This process of expanding child nodes, and alternating between evaluation and partial instantiation continues. A node is a leaf node if its true and false set of atoms cannot be unified. For our example, the partial instantiation tree is finite and has 11 nodes in total.

2.2 Review: Algorithm *SizeOpt*

The following algorithm intends to reduce the size of a given program by deleting clauses whose bodies cannot possibly be satisfied. Since as far as minimal model computation is concerned, a negative literal in the body of a clause can be moved to become a positive literal in the head, hereafter we only consider clauses possibly with disjunctive heads, but no negation in the bodies.

Algorithm SizeOpt ([4]) Input P, a ground disjunctive program, and S_0, the set of atoms that do not appear in the head of any clause in P.

1. Initialize Q to P, Q_d to $\emptyset$ and i to 0.

2. Set R to $\emptyset$.

3. For each clause $Cl \equiv A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$ in Q, and for some B_j such that $B_j \in S_i$

   (a) delete Cl from Q;

   (b) add Cl to Q_d; and

   (c) add $A_1, \ldots, A_m$ to R.

4. Increment i by 1, and set S_i to R.

5. For all A in S_i, if A occurs in the head of some clause in Q, delete A from S_i.

6. If S_i is empty, then return Q and Q_d, and halt. Otherwise, go back to Step 2. □

Hereafter, we use the notation $sizeopt(P) = \langle Q, Q_d \rangle$ to denote the application of the above algorithm on P, where Q is the set of retained clauses, and Q_d is the set of deleted clauses.

Example 1 Let P be the following program:

$$A \leftarrow B \wedge C \quad (1)$$
$$B \vee D \leftarrow A \wedge E \quad (2)$$
$$B \leftarrow E \wedge F \quad (3)$$
$$D \leftarrow A \quad (4)$$

Initially, S_0 is the set $\{C, E, F\}$. Thus, after Step 3 in the first iteration of Algorithm SizeOpt, Q_d consists of Clauses 1, 2 and 3, and the only clause remained in Q is Clause 4. After Step 5, S_1 is $\{A, B\}$. In the second iteration of Algorithm SizeOpt, the clause $D \leftarrow A$ is deleted from Q and added to Q_d in Step 3. S_2 is the set $\{D\}$. In the third iteration of Algorithm SizeOpt, execution halts as Q becomes empty. □

Example 2 Let P' be the program obtained by adding the following two clauses to P introduced in the previous example:

$$C \vee G \leftarrow \quad (5)$$
$$E \leftarrow C \quad (6)$$

When Algorithm SizeOpt is applied to P', the situation changes drastically. S_0 is now $\{F\}$. In the first iteration, Clause 3 is the only clause added to Q_d, and S_1 is empty after Step 5. Thus, the algorithm halts in Step 6 without another iteration. □

The above example demonstrates that Algorithm SizeOpt is not monotonic, i.e. $P_1 \subseteq P_2 \not\Rightarrow Q_{d,1} \subseteq Q_{d,2}$ where $sizeopt(P_1) = \langle \_, Q_{d,1} \rangle$ and $sizeopt(P_2) = \langle \_, Q_{d,2} \rangle$. It is also easy to see that Algorithm SizeOpt is not anti-monotonic either (i.e. $P_1 \subseteq P_2 \not\Rightarrow Q_{d,2} \subseteq Q_{d,1}$). The following lemma, proved in [4], shows that Algorithm SizeOpt preserves minimal models.

Lemma 1 ([4]) Let P be a disjunctive deductive database such that $sizeopt(P) = \langle Q, Q_d \rangle$. M is a minimal model of P iff M is a minimal model of Q. □

3 An Incremental Algorithm

Suppose P is the program considered in a node N of a partial instantiation tree, and $\theta_1, \ldots, \theta_m$ are all the conflict-set unifiers. As discussed in Section 2.1, Node N has m children, the j-th of which processes the instantiated program $P \cup P\theta_j$ (where $1 \leq j \leq m$). As described above, Algorithm SizeOpt can be applied to $P \cup P\theta_j$ to reduce the number of clauses that need to be processed. However, this approach of applying Algorithm SizeOpt directly may lead to a lot of repeated computations, as Algorithm SizeOpt has already been applied to P in Node N (and similarly, the programs in the ancestors of N). To avoid repetitive computations as much as possible, we develop Algorithm Incr that reuses $sizeopt(P)$ to produce $sizeopt(P \cup P\theta_j)$, as shown in Figure 2.

$$
\begin{tikzcd}
P \arrow[r] \arrow[d, "\text{Algorithm SizeOpt}"'] & P \cup P\theta_j \arrow[d, "\text{Algorithm SizeOpt}"] \\
sizeopt(P) \arrow[r, "\text{Algorithm Incr}"'] & sizeopt(P \cup P\theta_j)
\end{tikzcd}
$$

Figure 2: Incremental Maintenance

3.1 Graphs for Maintaining Deleted Clauses

Recall from Section 2 that $sizeopt(P)$ produces the pair $\langle Q, Q_d \rangle$, where Q_d consists of clauses deleted from P. To facilitate incremental processing, Algorithm Incr uses a directed graph G, called a *DC-graph*, to organize the deleted clauses. The intended properties of a DC-graph are as follows.

- Nodes represent atoms that do not appear in the head of any clause in Q.
- If there is an arc from node B_i to A, then the arc is labeled by a clause $Cl \in Q_d$ such that A appears in the head of Cl and B_i occurs in the body of Cl.

The only exceptions to the above properties are the special *root* node and the arcs originated from this root node. As will be shown later, the root node is the place where a graph traversal begins. Arcs originated from the root node are not labeled, as those arcs do not correspond to any clause in Q_d.

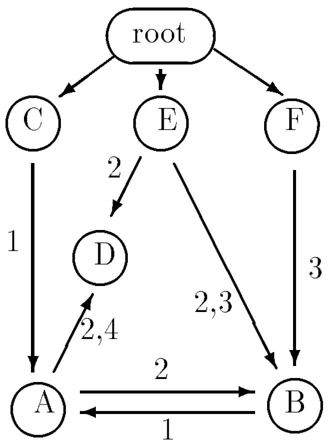

Figure 3: DC-graph G_1

Example 3 Consider the program P discussed in Example 1. Q_d consists of all 4 clauses in P. Figure 3 shows the DC-graph G_1 corresponding to Q_d. For convenience, arcs are labeled by the clause numbers used in Example 1. Furthermore, the label 2,3 of the arc from E to B is a shorthand notation that represents two arcs from E to B with labels 2 and 3 respectively. Notice that G_1 contains a cycle between A and B. $\square$

This example only illustrates how DC-graph G_1 looks like. We will show in Example 4 how G_1 can be constructed, after Algorithm Incr has been presented. However, before we can present the algorithm, we need the following concept.

3.2 Self-sustaining Cycles

Definition 1 Let $A_1 \xrightarrow{Cl_1} A_2 \xrightarrow{Cl_2} \dots \xrightarrow{Cl_{i-1}} A_i \dots A_n \xrightarrow{Cl_n} A_1$ be a cycle in DC-graph G, where $A \xrightarrow{Cl} B$ denotes an arc from A to B with label Cl. If there does not exist any arc from outside the cycle to some A_i with label Cl_{i-1} (i.e. $\nexists B \xrightarrow{Cl_{i-1}} A_i$ for some $B \notin \{A_1, \dots, A_n\}$), then the cycle is called *self-sustaining*. $\square$

As shown in the above example, G_1 contains the cycle $A \xrightarrow{Cl_2} B \xrightarrow{Cl_1} A$. This cycle is not self-sustaining because of the arc $C \xrightarrow{Cl_1} A$ (or the arc $E \xrightarrow{Cl_2} B$). The existence of this arc justifies why Clause 1 should be deleted, and why A should remain a node in the graph. On the other hand, if the arcs $C \xrightarrow{Cl_1} A$ and $E \xrightarrow{Cl_2} B$ were removed, the cycle $A \xrightarrow{Cl_2} B \xrightarrow{Cl_1} A$ became self-sustaining. Then for the sake of achieving the kind of incrementality depicted in Figure 2, Clause 2 should be restored (i.e. no longer be kept in Q_d). This would cause node B to disappear from the graph, which in turn leads to the restoration of Clause 1 and the disappearance of node A. Example 5 below will give further details as to why all these actions are necessary. In general, if there exists a self-sustaining cycle in a DC-graph, all the clauses involved in the cycle need to be restored, and all the nodes of the cycle need to be removed. We are now in a position to present Algorithm Incr.

3.3 Algorithm Incr

Algorithm Incr Input $P = \langle Q, Q_d \rangle$, the DC-graph G corresponding to Q_d, and a clause $Cl \equiv A_1 \vee \dots \vee A_m \leftarrow B_1 \wedge \dots \wedge B_n$ to be added to P.

1. For each B_i that does not appear in Q and Q_d (i.e. appearing in P the first time), add to graph G a node B_i and an arc from the root to node B_i.

2. For each B_i that is a node,

   (a) For each A_j where $1 \leq j \leq m$,

      (i) If A_j does not appear in Q, Q_d and G, add node A_j to G.

      (ii) If there is a node A_j in G, add an arc from node B_i to node A_j labeled Cl. If there is originally an arc from the root to node A_j, remove that arc.

(b) Add Cl to Q_d.

3. If there is no such B_i in the previous step,

   (a) Add Cl to Q.

   (b) For each A_j that appears as a node in G where $1 \leq j \leq m$, call Subroutine Remove(A_j).

4. For each self-sustaining cycle in G, call Subroutine Remove(D), where D is some atom in the cycle. $\square$

Subroutine Remove Input atom (node) A.

1. Remove from graph G node A and all the arcs pointing to A.

2. For each arc initially originating from A in G (i.e. $A \xrightarrow{Cl} B$),

   (a) Remove the arc from G.

   (b) If there does not exist another arc pointing to B with label Cl (i.e. $\not\exists D \xrightarrow{Cl} B$ for some D),

      (i) Remove Cl from Q_d, and add it to Q.

      (ii) Call Subroutine Remove(B) recursively.

3. For each clause Cl in Q_d such that A appears in the body of Cl, if all atoms in the body of Cl do not appear as nodes in G, remove Cl from Q_d, and add it to Q. $\square$

Hereafter, we use the notation $incr(\langle Q, Q_d, G \rangle, Cl) = \langle Q^{out}, Q_d^{out}, G^{out} \rangle$ to denote the fact that when Algorithm Incr is applied to inputs Q (the original set of retained clauses), Q_d (the original set of delete clauses), G (the DC-graph corresponding to Q_d), and Cl (the clause to be inserted), the outputs are Q^{out} (new set of retained clauses), Q_d^{out} (new set of deleted clauses), and G^{out} (new DC-graph). Moreover, we abuse notation by using $\emptyset$ to denote an empty DC-graph, i.e. the DC-graph with the root node only.

Example 4 Apply Algorithm Incr to the 4 clauses in the program P discussed in Example 1. In Figure 4, the first DC-graph (labeled (i)) is graph Gr_1 where $incr(\langle \emptyset, \emptyset, \emptyset \rangle, Cl_1) = \langle \emptyset, \{Cl_1\}, Gr_1 \rangle$. This is the case because nodes B and C are added in Step 1 of Algorithm Incr, node A and the two arcs pointing to A are added in Step 2a. Steps 3 and 4 are not needed in this case.

Similarly, the second graph in Figure 4 is DC-graph Gr_2 where $incr(\langle \emptyset, \{Cl_1\}, Gr_1 \rangle, Cl_2) = \langle \emptyset, \{Cl_1, Cl_2\}, Gr_2 \rangle$. This time, node E is added in Step 1 of Algorithm Incr, and the four arcs pointing from A and E to B and D are added in Step 2a. Notice that even though there is a cycle in Gr_2, the cycle is not self-sustaining. It is also not difficult to verify that $sizeopt(\{Cl_1, Cl_2\}) = \langle \emptyset, \{Cl_1, Cl_2\} \rangle$.

Similarly, the third graph in Figure 4 is produced by applying Algorithm Incr to add Cl_3 to Gr_2, and the fourth one (called G_1 in Example 3) is produced by applying Incr to

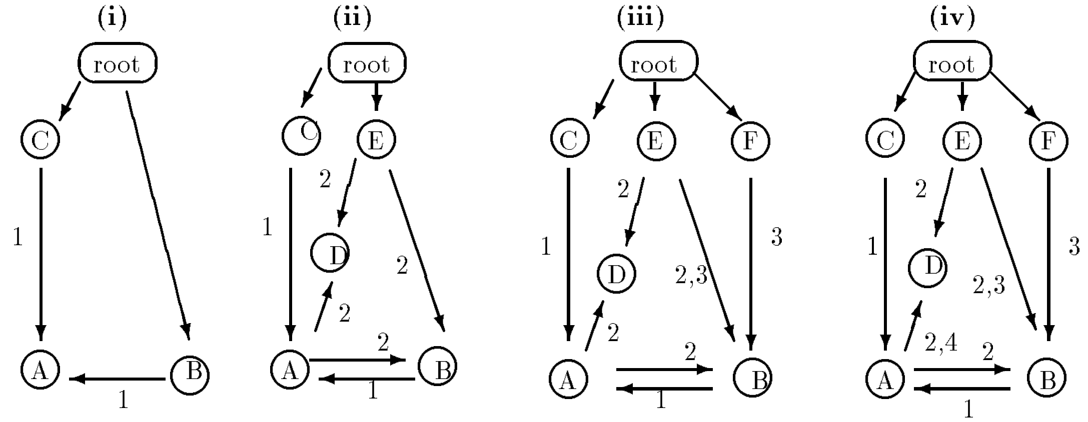

Figure 4: Applying Algorithm Incr to Add Clauses 1, 2, 3 and 4

Cl_4 and the third graph. Finally, the graphs in Figure 5 show the DC-graphs obtained by applying Algorithm Incr to insert the 4 clauses in the reverse order. As expected, the fourth DC-graphs in Figure 4 and Figure 5 are the same. Later we will show that inserting the clauses in different orders give identical end result. □

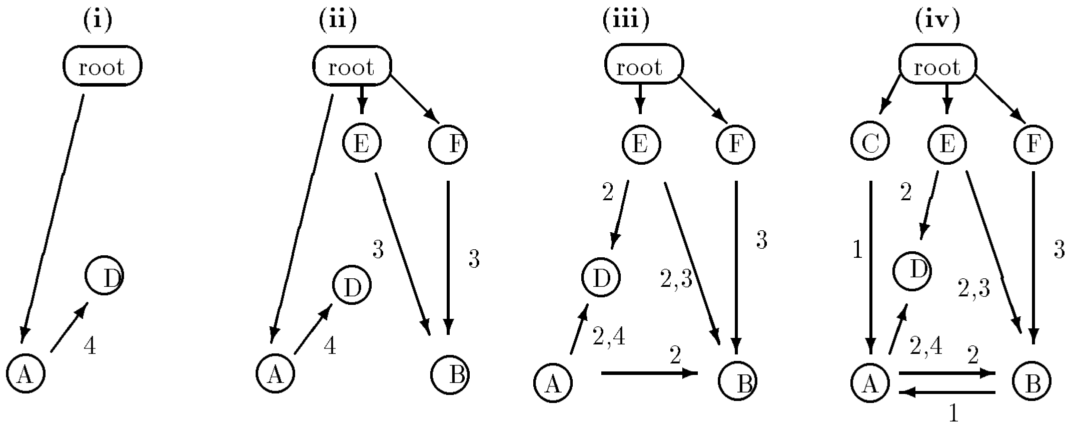

Figure 5: Applying Algorithm Incr to Add Clauses 4, 3, 2 and 1

The above example only demonstrates the situation when an inserted clause ends up being added to the set Q_d (i.e. Q_d keeps growing). Obviously, this is not always the case, as an inserted clause may indeed end up being added to the set Q. This addition may trigger a series of node removals and the shrinkage of Q_d.

Example 5 Now consider program P', that is by adding Clauses 5 and 6 discussed in Example 2. Let us add Clause 5 first. Steps 1 and 2 of Algorithm Incr are not invoked. But in Step 3a, the clause is added to Q, and Subroutine Remove(C) is called. In Step 1 of Subroutine Remove, node C and the arc from the root to C are removed. As for the arc from C to A labeled Cl_1, this arc is removed. But because of the existence of the arc from B to A labeled Cl_1, Subroutine Remove is *not* called recursively. Furthermore, Step 3 of Remove does not cause any change, and control returns to Algorithm Incr. As for Step 4 of Algorithm Incr, even though there is a cycle from between A and B, this cycle is not self-sustaining because of the arc from E to B with label Cl_2. Thus, Algorithm Incr halts. In functional terms, we have $incr(\langle\emptyset, \{Cl_1, \ldots, Cl_4\}, G_1\rangle, Cl_5) = \langle\{Cl_5\}, \{Cl_1, \ldots, Cl_4\}, Gr_5\rangle$, where Gr_5 is the first DC-graph shown in Figure 6. Before we proceed, note that it is not difficult to verify that $sizeopt(\{Cl_1, \ldots, Cl_5\}) = \langle\{Cl_5\}, \{Cl_1, \ldots, Cl_4\}\rangle$.

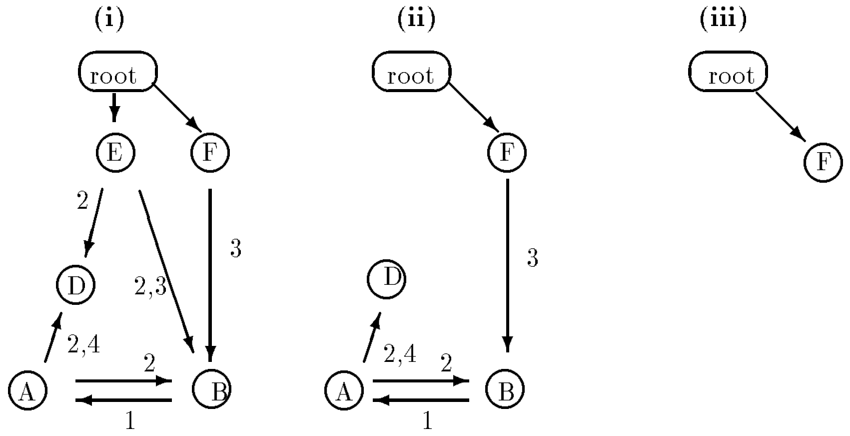

Figure 6: Applying Algorithm Incr to Add Clauses 5 and 6

Now let us add Clause 6. Steps 1 and 2 of Algorithm Incr are not invoked. But in Step 3a, the clause is added to Q, and Subroutine Remove(E) is called. In Step 1 of Subroutine Remove, node E and the arc from the root to E are removed. As for the arc from E to B labeled Cl_2, this arc is removed. But because of the existence of the arc from A to B labeled Cl_2, Subroutine Remove is *not* called recursively. Similarly, the arc from E to B labeled Cl_3 and the arc from E to D labeled Cl_2 are deleted without recursively calling Remove. Furthermore, Step 3 of Remove does not cause any change, and control returns to Algorithm Incr. The second DC-graph in Figure 6 shows the situation at this point.

However, unlike the above situation for Clause 5, this time the cycle between A and B is self-sustaining. Thus, in Step 4 of Algorithm Incr, Subroutine Remove(B) is called [2]. Step 1 of Remove(B) causes node B and the two arcs from F and A to B to be deleted. In Step 2, the arc from B to A is also removed; Clause 1 is moved from Q_d to Q; and this time

[2]It is not difficult to verify that the result is the same, if Remove(A) is called first.

Subroutine Remove(A) is invoked recursively. In Step 1 of Remove(A), node A is erased. In Step 2, the arc from A to D is removed; Clauses 2 and 4 are moved from Q_d to Q; and Subroutine Remove(D) is called recursively.

Step 1 of Remove(D) erases node D, and Step 3 causes no change. Control now returns to Step 3 of Remove(A). As there is no longer any clause in Q_d with A in the body, control returns to Step 3 of Remove(B). Again as there is no longer any clause in Q_d with B in the body, the executions of Remove(B) and Algorithm Incr are now completed. In functional terms, we have $incr(\langle\{Cl_5\}, \{Cl_1, \ldots, Cl_4\}, Gr_5\rangle, Cl_6) = \langle\{Cl_1, Cl_2, Cl_4, Cl_5, Cl_6\}, \{Cl_3\}, Gr_6\rangle$, where Gr_6 is the last DC-graph shown in Figure 6.

As shown in Example 2, we have $sizeopt(\{Cl_1, \ldots, Cl_6\}) = \langle\{Cl_1, Cl_2, Cl_4, Cl_5, Cl_6\}, \{Cl_3\}\rangle$, verifying once again the incremental nature of Algorithm Incr. As detailed above, this is due largely to Step 4, without which the final situation would be as shown in the second DC-graph of Figure 6, but not as in the third graph. □

Example 6 Thus far, we have not seen a situation in which Step 3 of Subroutine Remove is needed. But given the third graph in Figure 6, let us consider adding the clause $F \leftarrow A$ to the existing program. Since A appears in Q, Step 3 of Algorithm Incr adds the clause to Q and calls Remove(F). Now in Step 3 of Remove(F), Clause 3 – which is in Q_d, but does not appear as a label in G – is correctly inserted into Q from Q_d. □

3.4 Correctness Proof: Incrementality of Algorithm Incr

In the remainder of this section, we present one of the key results of this paper – the theorem proving the incremental property of Algorithm Incr (cf. Theorem 1). This property has been verified several times in the previous examples. But before we can prove the theorem, we need the following lemmas.

Lemma 2 Let P be the set $\{Cl_1, \ldots, Cl_n\}$. Then:

1. Let $sizeopt(P) = \langle Q, Q_d\rangle$. It is the case that $Q \cup Q_d = P$ and $Q \cap Q_d = \emptyset$.

2. Let $incr(\ldots incr(\langle\emptyset, \emptyset, \emptyset\rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n\rangle$. It is the case that $P_n \cup P_{n,d} = P$ and $P_n \cap P_{n,d} = \emptyset$.

Proof Outline For Part 1, as shown in Algorithm SizeOpt, Q is initialized to P, and Q_d to $\emptyset$ in Step 1. Afterwards, the only place where a clause is removed is in Step 3. More specifically, as shown in Steps 3a and 3b, whenever a clause is removed from Q, that clause is added to Q_d. Thus, it is obvious that Part 1 of the lemma is true.

For Part 2, let us prove by induction on n. When $n = 1$, it is obvious that Subroutine Remove is not invoked in Algorithm Incr. If Cl_1 is of the form $A_1 \vee \ldots \vee A_m \leftarrow$, then by Step 3, $P_1 = \{Cl_1\}$ and $P_{1,d} = \emptyset$. Otherwise, Cl_1 is of the form $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_u$. Then by Step 2, $P_1 = \emptyset$ and $P_{1,d} = \{Cl_1\}$. Hence, in both cases, $P_1 \cup P_{1,d} = \{Cl_1\}$ and $P_1 \cap P_{1,d} = \emptyset$.

Now assume that Part 2 of the lemma is true for $n = k-1$. There are two cases. First, consider the case when Subroutine Remove is not called. Then Steps 2 and 3 are the only places when a clause is either added to P_k or $P_{k,d}$. Notice that the conditions of Steps 2 and 3 are mutually exclusive to each other. Thus, given the induction assumption that $P_{k-1} \cup P_{k-1,d} = \{Cl_1, \ldots, Cl_{k-1}\}$ and $P_{k-1} \cap P_{k-1,d} = \emptyset$, it is the case that $P_k \cup P_{k,d} = \{Cl_1, \ldots, Cl_k\}$ and $P_k \cap P_{k,d} = \emptyset$.

Second, consider the case when Subroutine Remove is invoked. The two places in Remove when a clause is moved around are Steps 2a and 3. More specifically, whenever a clause is deleted from P_{k-1}, it is immediately added to P_k. Thus given the induction assumption, it is necessary that regardless of how many times Remove is invoked, $P_k \cup P_{k,d} = \{Cl_1, \ldots, Cl_k\}$ and $P_k \cap P_{k,d} = \emptyset$. □

The lemma above shows that for both Algorithm SizeOpt and Algorithm Incr, the set of retained clauses and the set of deleted clauses partition the original program P. The lemma below shows that node A appears in a DC-graph if and only if all clauses with A in the heads have already been deleted.

Lemma 3 Let $incr(\ldots incr(\langle\emptyset, \emptyset, \emptyset\rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n\rangle$. Then for any atom A, A appears as a node in G_n iff there does not exist any clause in P_n with A in the head.

Proof Outline Prove by induction on n. When $n = 1$, it is obvious that Subroutine Remove is not invoked in Algorithm Incr. If node A appears in the DC-graph, the node must be added in Step 2a. Then by Step 2c, Cl_1 is added to $P_{1,d}$, and is not in P_1. Conversely, if Cl_1 appears in P_1, then it must be added to P_1 in Step 3a. In that case, Step 2a is not executed, and A does not appear in the DC-graph. Now assume that the lemma is true for $n = k-1$. There are 2 cases.

<u>Case 1</u> Subroutine Remove is not called.

For any atom A, there are two subcases.

<u>Case 1.1</u> A does not appear in the head of Cl_k.

If A does not appear in the body of Cl_k, then A appears as a node in G_k iff A appears as a node in G_{k-1}, as Subroutine Remove is not invoked. By the induction assumption, A appears in G_k iff there does not exist any clause in P_{k-1} with A in the head. Since A is not the head of Cl_k, it is necessary that there does not exist any clause in P_k with A in the head.

Now consider the case when A appears in the body of Cl_k. If A appears in either P_{k-1} or $P_{k-1,d}$, then A appears as a node in G_k iff A appears as a node in G_{k-1}. The situation is exactly the same as the one considered in the previous paragraph. Otherwise, if A appears for the first time, then node A is added to G_k in Step 1. But obviously P_k still does not contain any clause with A in the head.

<u>Case 1.2</u> A appears in the head of Cl_k.

There are two subcases, depending on whether Step 2 or 3 is executed. If Step 3 is executed, then Cl_k is in P_k by Step 3a. But then Step 3b guarantees that G_k does not contain node A. On the other hand, if Step 2 is executed instead, there are two more subcases. If A appears in either P_{k-1} or $P_{k-1,d}$, then A appears as a node in G_k iff A appears as a node in G_{k-1}. The situation is then similar to the one considered in the first paragraph of Case 1.1.

Otherwise, if A appears for the first time, then node A is added to G_k in Step 2a. But then Cl_k is added to $P_{k,d}$ in Step 2b, but not added to P_k. By the induction assumption, since node A does not appear in G_{k-1}, there is no clause in P_{k-1} with A in the head. Thus, as Cl_k is added to $P_{k,d}$, there is no clause in P_k with A in the head. This completes the analysis of Case 1.
Case 2 Subroutine Remove is invoked.
For any atom A, there are two subcases.
Case 2.1 Remove(A) is invoked.
There are three places where Remove(A) can be invoked. If Remove(A) is called from Step 3b of Incr, then in Step 3a a clause with A in the head is added to P_k. If Remove(A) is called recursively in Step 2b of Remove(B) for some B, $B \xrightarrow{Cl} A$ is the only arc pointing to A with label Cl for some clause Cl with A in the head. Then in Step 2b of Remove(B), Cl is moved from $P_{k-1,d}$ to P_k. Finally, if Remove(A) is called from Step 4 of Algorithm Incr, A is in a self-sustaining cycle. Step 2 of Remove(A) recursively causes all nodes in the self-sustaining cycle be removed. Thus, at least one clause with A in the head is moved from $P_{k-1,d}$ to P_k.
Case 2.2 Remove(A) is not invoked.
The analysis for this case is very similar to the one for Case 1. This completes the proof of this lemma. □

We need one more lemma before we can prove Theorem 1. This lemma requires the following concept.

Definition 2 Let A be a node in a DC-graph G. The *rank* of A in G, denoted by $rank(A)$, is defined recursively as follows:

1. If there is an arc from the root to A, $rank(A) = 0$.

2. Let $B_{1,1}, \ldots, B_{1,u_1}, \ldots, B_{m,1}, \ldots, B_{m,u_m}$ be all the nodes that have arcs pointing to A, such that: a) $\{Cl_1, \ldots, Cl_m\}$ are all the labels of these arcs, and b) for all $1 \leq j \leq m$, $B_{j,1}, \ldots, B_{j,u_j}$ are all the nodes that have arcs pointing to A with label Cl_j. Then $rank(A) = 1 + max_{j=1}^{m}(min_{i=1}^{u_j}\, rank(B_{j,i}))$. □

Example 7 Consider the DC-graph G_1 introduced in Figure 3. The nodes with $rank = 0$ are C, E and F. Now consider $rank(A)$. There are the arcs from C and B pointing to A, both with label Cl_1. Thus, $rank(A) = 1 + min\{rank(C), rank(B)\}$. Since $rank(C) = 0$, it is obvious that $rank(A) = 1 + rank(C) = 1$. Now consider $rank(B)$ and all the arcs pointing to B. This time there are two different labels: Cl_2 and Cl_3. For Cl_2, there are the arcs from A and E to B. Based on an analysis similarly to the one for $rank(A)$, the minimum corresponding to Cl_2 is $rank(E) = 0$. For Cl_3, there are the arcs from E and F to B. Thus, the minimum based on Cl_3 is $min\{rank(E), rank(F)\} = 0$. Hence, $rank(B) = 1 + max\{0, 0\} = 1$, where the two zeros correspond to Cl_2 and Cl_3 respectively. Similarly, it is not difficult to verify that $rank(D) = 1 + rank(A) = 2$. Now compare the ranks with the sets S_0, S_1 and S_2 discussed in Example 1. The interesting thing here is that for all atoms A, $rank(A) = k$ iff $A \in S_k$. This property will be proved formally in the lemma below. □

Notice that if a DC-graph contains a self-sustaining cycle, rank assignments to atoms in the cycle are not well-defined. For example, consider the self-sustaining cycle between A and B in the second DC-graph in Figure 6. Then $rank(B)$ depends on $rank(A)$ which in turn depends on $rank(B)$. Thus, both ranks are not well-defined because of the cyclic dependency. Fortunately, since Step 4 of Algorithm Incr removes all self-sustaining cycles, all DC-graphs produced by Incr do not contain any self-sustaining cycle. Then by Definition 1, for the non self-sustaining cycle $A_1 \xrightarrow{Cl_1} A_2 \xrightarrow{Cl_2} \ldots \xrightarrow{Cl_{i-1}} A_i \ldots A_n \xrightarrow{Cl_n} A_1$, there must exist atom A_i such that there exists arc $B \xrightarrow{Cl_{i-1}} A_i$ for some atom $B \notin \{A_1, \ldots, A_n\}$. Thus, in determining $rank(A_i)$, for Clause Cl_{i-1}, $min\{rank(B), rank(A_{i-1})\}$ is always well-defined (cf. the previous example). Thus, there is no cyclic dependency on rank assignments.

Lemma 4 Let $incr(\langle Q, Q_d, G\rangle, Cl) = \langle Q^{out}, Q_d^{out}, G^{out}\rangle$. Then for all nodes $A \in G^{out}$, $rank(A) = n$ iff $A \in S_n$, where the sets $S_0, \ldots, S_n, \ldots$ are the ones produced by applying Algorithm SizeOpt directly on $Q^{out} \cup Q_d^{out}$.

Proof Outline Prove by induction on n. When $n = 0$, $rank(A) = 0$ iff there is an arc from the root to A. This arc is created in Step 1 of Algorithm Incr. If this arc is not removed in Step 2b, it must be the case that A does not appear in the head of any clause in $Q^{out} \cup Q_d^{out}$. Then when applying Algorithm SizeOpt directly on $Q^{out} \cup Q_d^{out}$, it is necessary that $A \in S_0$. Assume that the lemma is true for $n = k - 1$. We prove the if and only-if part separately.

<u>Case 1</u> $rank(A) = k$

By Definition 2, $rank(A) = 1 + max_{j=1}^{m}(min_{v=1}^{u_j} rank(B_{j,v}))$. That is, among the clauses $Cl_1, \ldots, Cl_m$ that are the labels of all the arcs pointing to A, there exists one clause Cl_j where $1 \leq j \leq m$ such that $rank(A) = k = 1 + (min_{v=1}^{u_j} rank(B_{j,v}))$. More specifically, Cl_j must be of the form $\ldots A \ldots \leftarrow \ldots \wedge B_{j,1} \wedge \ldots \wedge B_{j,u_j} \wedge \ldots$. Among these u_j atoms, let i be the one so that $rank(B_{j,i}) = min_{v=1}^{u_j} rank(B_{j,v})$. In other words, $rank(B_{j,i}) = k - 1$. By the induction assumption, $B_{j,i} \in S_{k-1}$. Thus, in Step 3 of Algorithm SizeOpt, Cl_j is removed, and A is added to the set R. By applying a similar argument, it is obvious that all clauses $Cl_1, \ldots, Cl_m$ must be removed at some iteration of Algorithm SizeOpt. More specifically, since Cl_j corresponds to the maximum "minimum-rank", Cl_j must be the last clause deleted with A appearing in the head. Thus, there must not exist any retained clause with head A. Hence, in Step 5 of Algorithm SizeOpt, A is kept in the set S_k.

<u>Case 2</u> $A \in S_k$

As shown in Algorithm SizeOpt, there must exist a clause Cl_j of the form $\ldots A \ldots \leftarrow \ldots B_{j,i} \ldots$, such that this is (one of) the last clause with A in the head, and $B_{j,i}$ is in S_{k-1}. By the induction assumption, $rank(B_{j,i}) = k - 1$. Now among all $B_{j,1}, \ldots, B_{j,u_j}$ that appear in the body of Cl_j and that appear as nodes in the DC-graph, suppose there exists $B_{j,l}$ such that $rank(B_{j,l}) < k - 1$. By the induction assumption, $B_{j,l} \in S_w$ where $w < k - 1$. In that case, by Step 3 of Algorithm SizeOpt, the clause Cl_j must have been deleted earlier, and should not exist for deletion in the current iteration. This is a contradiction. Thus, it is necessary that $rank(B_{j,i}) = min_{v=1}^{u_j} rank(B_{j,v})$. By applying a similar argument, for every clause Cl_w among $Cl_1, \ldots, Cl_m$ with A in the heads, there exists an i_w for $1 \leq w \leq m$ such that $rank(B_{w,i_w}) = min_{v=1}^{u_w} rank(B_{w,v})$. But since Cl_j is the last clause to be deleted, it is necessary that $rank(B_{j,i}) = rank(B_{j,i_j}) = max\{B_{1,i_1}, \ldots, B_{m,i_m}\}$. Hence, it is necessary that $rank(A) = 1 + rank(B_{j,i}) = k$. □

Now we are in a position to present the theorem that proves the incremental property of Algorithm Incr.

Theorem 1 Let P be a program consisting of clauses $Cl_1, \ldots, Cl_n$. Let $sizeopt(P) = \langle Q, Q_d \rangle$, and let $incr(\ldots incr(\langle \emptyset, \emptyset, \emptyset \rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n \rangle$. Then: $Q = P_n$ and $Q_d = P_{n,d}$.

Proof Outline Given Lemma 2, it suffices to prove $Q_d = P_{n,d}$. Let $Cl \equiv \ldots A \ldots \leftarrow B_1 \wedge \ldots \wedge B_m$ be a clause in Q_d.

Case 1 No clause in Q with A in the head

Then all clauses with A in the head are in Q_d, and for some k, $A \in S_k$. By Lemma 4, this is true iff $rank(A) = k$. By Lemma 3, this is possible iff all clauses with A in the heads have been deleted, i.e. in $P_{n,d}$.

Case 2 exists some clause in Q with A in the head

Cl is in Q_d iff there exists B_j where $1 \leq j \leq m$ such that $B_j \in S_k$ for some k. By Lemma 4, this is true iff $rank(B_j) = k$. There are now two subcases depending on whether node B_j appears in the DC-graph when Cl was inserted by Algorithm Incr.

Case 2.1 Node B_j already created

Then by Step 2c of Algorithm Incr, Cl is added to the set of deleted clauses.

Case 2.2 Otherwise

Suppose Cl does not represent the first time B_j appears. Let Cl_1 be the clause when B_j first appears. Since there does not exist node B_j in the DC-graph, B_j must be in the head of Cl_1, as ensured by Step 1 of Algorithm Incr. Furthermore, because of Step 2, and because there does not exist node B_j in the graph, Cl_1 must be added to the set of retained clauses in Step 3. But notice that in Algorithm Incr and Subroutine Remove, once a clause is put into the set of retained clauses, it will never be removed. In other words, Cl_1 must be in P_n. However, by Lemma 3, B_j cannot be a node in the graph G_n, and $rank(B_j)$ cannot be equal to k. This is a contradiction. Hence, it is necessary that Cl represents the first time B_j appears. Thus, in Step 1 of Algorithm Incr, a node for B_j is created, and the situation is exactly the same as in Case 2.1.

Combining Cases 2.1 and 2.2, it is necessary that Cl was once added to the set of deleted clauses. Now since B_j is a node in the DC-graph, Step 3 of Subroutine Remove will never remove Cl from the set of deleted clauses. Hence, it is necessary that Cl is in $P_{n,d}$. This completes the proof of the theorem. □

Corollary 1 Given clauses $Cl_1, \ldots, Cl_n$, Algorithm Incr produces the same end result regardless of the order $Cl_1, \ldots, Cl_n$ are inserted. □

4 Further Optimizations

In the previous section, we have presented Algorithm Incr and showed that it achieves the kind of incrementality shown in Figure 2. In this section, we will develop several ways to optimize this algorithm, and the expansion and computation of an partial instantiation tree.

4.1 Algorithm IncrOpt

A complexity analysis on Algorithm Incr reveals that Step 4 plays a considerable role in determining the efficiency of Incr. It involves finding each and every self-sustaining cycle that may exist in the DC-graph. As shown in Example 5, this is the crucial step that leads to the incremental property of Algorithm Incr. However, the following lemma shows that from the point of view of computing minimal models, self-sustaining cycles need not be detected, and can be left in the graph.

Lemma 5 Let Q be a set of retained clauses and Q_d be a set of deleted clauses maintained in the DC-graph G. Let $A_1 \xrightarrow{Cl_1} A_2 \xrightarrow{Cl_2} \ldots \xrightarrow{Cl_i} A_{i+1} \ldots A_n \xrightarrow{Cl_n} A_1$ be a self-sustaining cycle in G. M is a minimal model of $Q \cup \{Cl_1, \ldots, Cl_n\}$ iff M is a minimal model of Q.

Proof Outline As introduced in Section 3.1, for all $1 \leq i \leq n$, Cl_i is a clause with A_{i+1} in the head and A_i in the body. Since $A_1, \ldots, A_n$ are nodes in DC-graph G, none of $A_1, \ldots, A_n$ appears in Q. Thus, given any minimal model M of Q, none of $A_1, \ldots, A_n$ is contained in M. Then it is easy to see that M is a model of $Cl_1, \ldots, Cl_n$. Hence, M is a minimal model of $Q \cup \{Cl_1, \ldots, Cl_n\}$ iff M is a minimal model of Q. $\square$

The above lemma motivates the following algorithm.

Algorithm IncrOpt Exactly the same as Algorithm Incr, but without Step 4 of Incr. $\square$

Hereafter we use the notation $incropt(\langle Q, Q_d, G\rangle, Cl) = \langle Q^{out}, Q_d^{out}, G^{out}\rangle$ for Algorithm IncrOpt in exactly the same way as we use $incr(\langle Q, Q_d, G\rangle, Cl) = \langle Q^{out}, Q_d^{out}, G^{out}\rangle$ for Incr. The corollary below follows directly from Lemma 1, Theorem 1 and Lemma 5.

Corollary 2 Let P be a program consisting of clauses $Cl_1, \ldots, Cl_n$, and let $incropt(\ldots incropt(\langle \emptyset, \emptyset, \emptyset\rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n\rangle$. M is a minimal model of P iff M is a minimal model of P_n. $\square$

As far as supporting minimal model computation is concerned, Algorithm IncrOpt is more preferable than Algorithm Incr. The reasons are threefold.

- First, as discussed above, IncrOpt does not check for self-sustaining cycles. While cycle detection takes time linear to the number to edges in the graph, checking *all* cycles to see whether they are self-sustaining takes considerably more time. Thus, by not checking self-sustaining cycles, IncrOpt is more efficient than Incr.

- Second, it is easy to see if $incropt(\langle Q, Q_d, G\rangle, Cl) = \langle Q_{opt}^{out}, \_, G_{opt}^{out}\rangle$ and $incr(\langle Q, Q_d, G\rangle, Cl) = \langle Q^{out}, \_, \_\rangle$, then it is necessary that $Q_{opt}^{out} \subseteq Q^{out}$. More precisely, IncrOpt keeps all clauses in self-sustaining cycles deleted. Thus, the size of the program Q_{opt}^{out} may be much smaller than that of Q^{out}. The implication is that finding the minimal models based on Q_{opt}^{out} may take considerably less time than finding the minimal models based on Q^{out}.

- The third reason why Algorithm IncrOpt is more preferred applies only to programs P that are definite (i.e. no disjunctive heads). The following lemma shows that for such programs P, Algorithm IncrOpt directly finds the least model of P.

Lemma 6 Let P be a definite program consisting of clauses $Cl_1, \ldots, Cl_n$, and let $incropt(\ldots incropt(\langle \emptyset, \emptyset, \emptyset \rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n \rangle$. The least model of P is the set $\{A | A$ is the head of a clause in $P_n\}$.

Proof Outline Prove by induction on n. When $n = 1$, if Cl_1 is of the form $A \leftarrow$, Step 3 of IncrOpt adds Cl_1 to P_1. Then it is obvious that the least model of Cl_1 is the set $\{A\}$. On the other hand, if if Cl_1 is of the form $A \leftarrow B_1 \wedge \ldots \wedge B_m$, Step 2 of IncrOpt adds Cl_1 to $P_{1,d}$, and P_1 becomes empty. Then it is easy to see that the least model of Cl_1 is the empty set. Now assume that the lemma is true for $n = k - 1$. There are two cases.
Case 1 Cl_k is added to $P_{k,d}$.
This must occur in Step 2 of IncrOpt, and Cl_k is of the form $A \leftarrow B_1 \wedge \ldots \wedge B_m$ such that there exists a B_j for $1 \leq j \leq m$ that appears as a node in the DC-graph G_k. There are two subcases. First, B_j may be added as a node in Step 1 of IncrOpt, in which case B_j appears for the first time and must not be in the least model of $Cl_1, \ldots, Cl_k$. Alternatively, B_j may be a node in DC-graph G_{k-1}. Then according to Lemma 3, B_j cannot be the head of a clause in P_{k-1}. By the induction assumption, B_j is not in the least model of $Cl_1, \ldots, Cl_{k-1}$, and hence not in the least model of $Cl_1, \ldots, Cl_k$. By combining the two subcases, it is necessary that the least model of $Cl_1, \ldots, Cl_k$ is the same as the least model of $Cl_1, \ldots, Cl_{k-1}$. By the induction assumption, the latter is the set $\{A | A$ is the head of a clause in $P_{k-1}\}$. But since Cl_k is added to $P_{k,d}$, it is necessary that $P_k = P_{k-1}$.
Case 2 Cl_k is added to P_k.
Let Cl_k be of the form $A \leftarrow B_1 \wedge \ldots \wedge B_m$. There are again two subcases depending on whether Subroutine Remove is invoked. First, consider the subcase when Remove is not called. Then $P_k = P_{k-1} \cup Cl_k$, and thus $\{B | B$ is the head of a clause in $P_k\}$ is equal to $\{A\} \cup \{B | B$ is the head of a clause in $P_{k-1}\}$. Moreover, Cl_k is added to P_k in Step 3 of IncrOpt. This is possible only if all B_j's do not occur as nodes in G_{k-1}. Then according to Lemma 3, all B_j's occur as heads of clauses in P_{k-1}. By the induction assumption, all B_j's are in the least model of $Cl_1, \ldots, Cl_{k-1}$. Thus, A is in the least model of $Cl_1, \ldots, Cl_k$.

Now consider the subcase when Subroutine Remove is called. A clause Cl may be added to P_k in Step 2b or 3 of Remove. If Cl is added in Step 2b, Cl is of the form $B \leftarrow A \wedge B_1 \wedge \ldots \wedge B_m$ where A occurs as the head of a clause in P_k, and thus is in the least model based on the analysis for the first subcase. Moreover, due to the condition of Step 2b, $B_1, \ldots, B_m$ must all be in the least model as well. Thus, B has to be in the least model. Alternatively, if Cl is added in Step 3 of Remove, this is possible only if all atoms in the body of Cl are not in the DC-graph, and are in the least model. Hence, the head of Cl must also be in the least model. □

The lemma above shows that when applying Algorithm IncrOpt to a definite program, once IncrOpt completes its execution, no further processing is needed to compute the least model.

This is not the case for Algorithm Incr and Algorithm SizeOpt, as shown in the following example.

Example 8 Consider the definite program $\{A \leftarrow B, B \leftarrow A, C \leftarrow\ , D \leftarrow C\}$. All 4 clauses remain if either Algorithm Incr or Algorithm SizeOpt is applied. The application of a least-model solver is then needed to compute the least model $\{C, D\}$. But if Algorithm IncrOpt is used instead, only the clauses $C \leftarrow$ and $D \leftarrow C$ remain, whose heads directly give the least model.

One may wonder whether the above lemma can be generalized to disjunctive programs in the following sense. If P is a disjunctive program consisting of clauses $Cl_1, \ldots, Cl_n$, and $incropt(\ldots incropt(\langle\emptyset, \emptyset, \emptyset\rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n\rangle$, then is it true that for all atoms A that appears in the head of a clause in P_n, A occurs in some minimal model of P? The answer is no. Consider $P = \{A \vee B \leftarrow\ , A \leftarrow\ , C \leftarrow B\}$. Applying IncrOpt does not cause any change. Thus, the set of atoms appearing in the heads is $\{A, B, C\}$. However, B and C are not contained in the (unique) minimal model of P. □

According to Corollary 1 and Lemma 5, when using Algorithm IncrOpt, different orders of inserting the same collection of clauses do not affect the final DC-graph, and the final sets of retained and deleted clauses. However, different orders may require different execution times – depending largely upon how many times Subroutine Remove is invoked. If Remove is not called at all when inserting a clause $A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_l$, the complexity of Algorithm IncrOpt is $O(ml)$. Otherwise, if a is the number of nodes (atoms) in the current graph, then the worst case complexity of recursively calling Remove is $O(alN)$, and that of IncrOpt is $O(ml + alN)$. It is then tempting to conclude that the complexity of IncrOpt for inserting n clauses is $O(n(ml + alN))$. However, this is incorrect because during the process of inserting the n clauses, Remove(A) for all atoms A can only occur at most once. Thus, for inserting n clauses, the complexity of IncrOpt should be $O(nml + al(N + n))$.

On the other hand, if Algorithm SizeOpt is used directly, then there are $(N+n)$ clauses [3]. The worst case complexity of Algorithm SizeOpt for $(N+n)$ clauses is $O(ml(N+n)^2)$. Thus, comparing the complexity figures of Algorithm SizeOpt and IncrOpt does not provide any clear conclusion, as the comparison depends on the magnitude of a, the number of atoms in a DC-graph, relative to the magnitudes of N, n, l and m. In Section 5, we will present experimental results evaluating the effectiveness of Algorithm IncrOpt.

4.2 Heuristics: Ordering Clauses to be Inserted

The above coarse-grained complexity analysis of Algorithm IncrOpt reveals that given n clauses to be inserted, the most efficient order is the one that minimizes the number of times Subroutine Remove needs to be called. In the following, we discuss three possible ways to insert n clauses. The most obvious way is to use IncrOpt to insert the clauses in an arbitrary order (e.g. textual order). For lack of a better name, we will refer to this strategy

[3]Based on Figure 2, the analysis here assumes that P consists of N clauses, and $P\theta_j$ consists of n clauses.

as IncrOptArb. To the other extreme, another way to insert n clauses is to really try to minimize the number of times Subroutine Remove will be called. The following algorithm uses a heuristic order that attempts to do that.

Algorithm IncrOptOrder Let $Cl_1, \ldots, Cl_n$ be the clauses to be inserted.

1. Initialize R to all the facts among $Cl_1, \ldots, Cl_n$, and S to $\emptyset$.
2. For each clause $Cl \in R$,
   (a) Call Algorithm IncrOpt with Cl.
   (b) If Cl is not added to the DC-graph, then for each atom A in the head of Cl, add all the clauses not considered so far with A in the body to S.
3. If S is not empty, set R to S and S to $\emptyset$. Go to Step 2.
4. Apply IncrOpt on each of the clauses not considered so far in an arbitrary order. □

Example 9 Suppose the six clauses of P and P' in Examples 1 and 2 are to be inserted. Clause 5 is the first one considered. Since IncrOpt does not add Clause 5 to the DC-graph, Clauses 1 and 6 are added to the set S and inserted in the next iteration of IncrOptOrder. While Clause 1 is added to the DC-graph, Clause 6 is not, which causes Clauses 2 and 3 to be considered in the third iteration. This time both clauses are added to the DC-graph. Then Step 4 of IncrOptOrder applies IncrOpt to Clause 4, the only clause remaining.

Notice that if Clause 5 is inserted after Clause 1, then node C created during the insertion of Clause 1 will need to be removed. Similarly, if Clause 6 is inserted after Clause 2, then node E will need to be removed. To prevent all these unnecessary insertions/removals from happening, IncrOptOrder inserts facts first and follows Step 2b. □

One possible weakness of Algorithm IncrOptOrder is that there may be too much overhead involved in implementing Step 2. The following algorithm represents a compromise. It inserts the facts among the n clauses first, but leaves the remaining clauses to be inserted in whatever order.

Algorithm IncrOptFact Let $Cl_1, \ldots, Cl_n$ be the clauses to be inserted. Apply Algorithm IncrOpt first to all the facts among the clauses. Then apply Algorithm IncrOpt to the remaining clauses in an arbitrary order. □

In Section 5, we will present experimental results evaluating the effectiveness of these three algorithms.

4.3 Avoiding Redundant Node Expansion

As described in Section 2.1, for each conflict-set unifier θ of a node in a partial instantiation tree, there is a child node processing $P \cup P\theta$. The lemma below attempts to reduce the time taken to expand a partial instantiation tree by not expanding those nodes that can be predicted to be identical to nodes that have already been generated. It gives 3 sufficient conditions which are very easy to implement. Without loss of generality, it assumes that substitutions in conflict-set unifiers are represented in solved form [19]. That is, for a set of (substitution) equations, the equations are of the form $X_j = t_j$, and all variables appearing in the left-hand-side of the equations cannot appear in the right-hand-side of any equation. For the following lemma, we use the notation $L(\theta)$ and $R(\theta)$ to denote the set of all variables appearing in the left-hand-side and right-hand-side of θ respectively. We also use the notation $P \overset{\theta}{\Rightarrow} P_1$ to denote the fact that the node for program P is the parent of the node for P_1, and θ is the conflict-set unifier, i.e. $P_1 = P \cup P\theta$.

Lemma 7

1. Given $P \overset{\theta}{\Rightarrow} P_1$ and $P_1 \overset{\theta}{\Rightarrow} P_2$, it is necessary that $P_2 = P_1$.
2. Given $P \overset{\theta_1}{\Rightarrow} P_1 \overset{\theta_2}{\Rightarrow} P_2$, and $P \overset{\theta_2}{\Rightarrow} P_3 \overset{\theta_1}{\Rightarrow} P_4$, $P_4 = P_2$ if: $L(\theta_1) \cap L(\theta_2) = \emptyset$, $L(\theta_1) \cap R(\theta_2) = \emptyset$, and $R(\theta_1) \cap L(\theta_2) = \emptyset$.
3. Given $P \overset{\theta_1}{\Rightarrow} P_1 \overset{\theta_2}{\Rightarrow} P_2 \overset{\theta_1}{\Rightarrow} P_3$, $P_3 = P_2$ if $L(\theta_1) \cap R(\theta_2) = \emptyset$.

Proof Outline For space considerations, we only show a proof outline for Part 3. By definition, $P_3 = P_2 \cup P_2\theta_1$. Substituting $P_2 = P_1 \cup P_1\theta_2$ into $(P_2 \cup P_2\theta_1)$, we get $P_1 \cup P_1\theta_2$ $\cup P_1\theta_1 \cup P_1\theta_2\theta_1$. Since $L(\theta_1) \cap R(\theta_2) = \emptyset$, $P_1\theta_2\theta_1 = P_1\theta_2$. Then it is straightforward to verify that by substituting $P_1 = P \cup P\theta_1$, $P_3 = P_2$. □

As an example, consider again the program P discussed in Section 2.1. As shown in Figure 1, P_2, which is defined by $P = P \cup P\theta_2$, has two child nodes corresponding to the conflict-set unifiers θ_1 and θ_2. Then according to the the first part of the above lemma, there is no need to expand the node $P_3 = P_2 \cup P_2\theta_2$, because P_3 is identical to P_2. And by the second part of the lemma, there is no need to expand the node P_6. In the next section, we will present experimental results showing the effectiveness of the optimizations described by the lemma.

5 Implementation Overview and Experimental Evaluation

In this section, we will present experimental results evaluating the effectiveness of the proposed algorithms and optimizations. But before we do that, we will first give an overview of the implementation of these algorithms and optimizations, as well as the implementation of the entire framework that includes both the evaluation and partial instantiation phases.

5.1 Implementation Overview of the Proposed Algorithms and Optimizations

For our experimentation, we implemented Algorithms IncrOpt (and thus trivially IncrOptArb), IncrOptOrder and IncrOptFact in C. We also implemented two versions of Algorithm

SizeOpt. One is a straightforward encoding of the algorithm presented in 2.2 in C. The other one tries to minimize searching by extensive indexing. Unfortunately, in all the experiments we have carried out so far, the version with extensive indexing requires so much overhead to set up the indices that the straightforward version takes much less time. Thus, for all the experimental results reported later for Algorithm SizeOpt, the straightforward version was used.

Recall that in our incremental algorithms, a DC-graph is used to organize the deleted clauses. Each arc in the graph represents a deleted clause. However, not every deleted clause has a corresponding arc in the graph. Given a deleted clause $Cl \equiv A_1 \vee \ldots \vee A_m \leftarrow B_1 \wedge \ldots \wedge B_n$, if all of $A_1, \ldots, A_m$ do not appear in the graph, then this clause would not appear as a label of an arc. In our implementation of the incremental algorithms, we set up a virtual node so that there is an arc from the appropriate node of an atom appearing in the body to the virtual node. More precisely, a virtual node is an atom that appears both in the heads of some clauses in Q and in the heads of some clauses in Q_d. In this way, each deleted clause has a corresponding arc in the DC-graph. This simplifies the construction and maintenance of DC-graph, and makes the implementation more efficient. This is because with the use of virtual nodes, Step 3 of Subroutine Remove can be skipped. Finally, to further speed up the maintenance of DC-graphs, a counter is kept for each clause which records the number of times the clause appears as an arc in the graph. If this counter decreases to zero, the clause is removed from Q_d, and put back to Q.

5.2 Implementation Overview of the Entire Framework

Apart from the proposed algorithms and optimizations, we also implemented the entire partial instantiation framework that given an input logic program, computes the entire partial instantiation tree. The entire system was written in C running under the UNIX environment, and has roughly 3000 lines of code. In the following, we summarize the main aspects of the implementation, and highlight how we tried to make the implementation as space and run-time efficient as possible.

5.2.1 Major Data Structures

There are four major data structures used in the system: a term table, an atom table, a clause table, and a partial instantiation tree structure. First, all the terms are organized in a global term table, in which each term is identified by an index. Associated with each term are such pieces of information as the type (i.e., constant, variable or function), arity, name, and pointers to the parameters of the term. At the root node of the partial instantiation tree, the term table only consists of those terms that are in the original program. When a child node is created, new terms generated via unification are added to the end of the term table. Note that when a child node and its subtree have been fully expanded, the part of the term table corresponding to the entire subtree can be thrown away. This leads to two implementation decisions. First, the expansion of a partial instantiation tree is conducted in a depth-first manner. Second, the term table is implemented as a stack. These decisions help to minimize the run-time space requirement of our system.

Every atom is stored in a global atom table which keeps track of such information as the name, arity, and the terms (represented by their indices to the term table) that appear in the atom. Like the term table, the atom table is organized as a stack. Similarly, there is a global clause table/stack which records for each clause the atoms appearing in the clause, in the form of indices to the atom table. Recall that when a child node is to be created, the program P at the parent node will be instantiated to $P \cup P\theta$. To facilitate the comparisons of the clauses in $P\theta$ with the existing clauses in P, atom indices in the clause table are kept in ascending order.

Last but certainly not the least, there is a partial instantiation tree structure. Apart from the usual parent and children pointers, each node has pointers to the set of unifiers, the true and false sets, and the appropriate DC-graphs. It also contains indices to the clause, atom and term tables. Again once a subtree has been fully expanded, as much space previously occupied by its nodes as possible is freed for future reuse.

5.2.2 Generation of New Clauses

Given a program P in a parent node and a conflict-set unifier θ, the program in the child node $P \cup P\theta$ is obtained by first getting all the appropriate unified terms $T\theta$. There are three possibilities for $T\theta$. It may be T itself, the same as some existing term in the term table, or an entirely new term. In the latest case, the new term is added to the term table, and a pointer from T to $T\theta$ is created. This kind of pointers will assist in the (possible) insertion of a new, unified atom $A\theta$ into the atom table. This insertion in turn creates a pointer from atom A to $A\theta$. Again this kind of pointers facilitates the insertion of unified clauses to the clause table.

It is obvious that in generating a child node, a lot of comparisons for terms, atoms and clauses need to be made. In particular, to check whether a term/atom/clause is new or not, it is compared with every term/atom/clause in the appropriate tables. Thus, our implementation of the tables as stacks does not only reduce run-time memory space requirement, but also minimizes the time taken for comparisons. Furthermore, as discussed above, comparisons are facilitated by keeping atom indices in ascending order in the clause table.

5.2.3 Unification

In partial instantiation, generating the conflict-set unifiers is a key step at each node. Thus, the efficiency of the unification algorithm is one of the key factors determining the overall performance of the system. Among the unification algorithms that have been proposed so far (e.g., [7, 19]), we chose to implement the version developed by Martelli and Montanari [19], with a few optimizations. For instance, a key optimization is to keep all the variables appearing in the left-hand-sides of substitution equations in sorted order. Thus, unifiers can be compared more efficiently.

In the remainder of this section, we will report experimental results evaluating the effectiveness of our proposed algorithms and optimizations. All run-times are in milliseconds, and were obtained by running the experiments in a SPARC-LX Unix time-sharing environment.

5.3 IncrOptFact vs IncrOptOrder vs IncrOptArb

In this series of experiments, we compared the effectiveness of the heuristics described in Section 4.2. The following results are very representative of all the experiments we conducted. The times below count the time taken for each algorithm to process 20 clauses. At most 5 atoms appear in the head of each clause, and at most 10 appear in the body. All atoms in the heads and bodies, as well as their numbers, are randomly generated.

	IncrOptFact	IncrOptArb	IncrOptOrder
time (ms)	3.5	3.6	150.6

Recall that IncrOptOrder tries to minimize the number of times Subroutine Remove needs to be called by first inserting the facts, and then partially ordering the insertion of the remaining clauses. Clearly shown above, the strategy backfires as it requires too much overhead. Inserting a set of clauses in arbitrary order, as shown in the third column of the above table, performs surprisingly well. However, IncrOptFact is considered to be the best, not so much because it outperforms IncrOptArb by a wide margin, but rather because it is very simple to implement, and almost always performs better than IncrOptArb. In the remainder of this section, we will only report the results of IncrOptFact.

5.4 Same Number of Disjunctive Clauses: IncrOptFact vs SizeOpt

In this series of experiments, we compared the effectiveness of our incremental algorithm IncrOptFact with the original algorithm SizeOpt. For each algorithm, we report i) the total time taken to process the 20 clauses used in Section 5.3, ii) the number of clauses deleted, and iii) the time taken to find the minimal models.

	IncrOptFact	SizeOpt
processing time for 20 clauses (ms)	3.54	0.33
rules deleted	19	0
time to find minimal models (ms)	49.17	83.61
total time taken (ms)	52.71	83.94

For just the time taken to process the 20 clauses, our incremental algorithm IncrOptFact takes more time than SizeOpt, primarily for maintaining DC-graphs. But as shown above, the extra time is worth spending because IncrOptFact manages to delete 19 more clauses than SizeOpt. This is all due to the fact that, as described in Section 4.1, IncrOptFact deletes all the clauses in self-sustaining cycles. Consequently, the times taken for the two algorithms to find the (same collection of) minimal models differ by a wide margin. This clearly demonstrates the importance of deleting more rules, whose impact is multiplied in model computations. At the end, the total time taken by IncrOptFact is only about 60% of the time taken by SizeOpt.

5.5 Same Number of Definite Clauses: IncrOptFact vs SizeOpt

Based on the results of the previous set of experiments for disjunctive clauses, we surely can predict that for definite clauses, IncrOptFact again outperforms SizeOpt. Moreover, Lemma 6 presents a stronger reason for us to believe that IncrOptFact will perform even better. The lemma shows that for definite clauses, our incremental algorithms can obtain the least model by simply obtaining the heads of all the clauses not deleted. Indeed, our belief is confirmed by this series of experiments, in which each test program contains 100 randomly generated definite clauses. The following table reports the run-times for a typical program.

	IncrOptFact	SizeOpt
processing time for 100 clauses (ms)	9.22	0.73
rules deleted	89	17
time to find least model (ms)	5.76	580.06
total time taken (ms)	14.98	580.79

The processing time taken by IncrOptFact is longer than that by SizeOpt. But again IncrOptFact deletes many more clauses, and requires a minimal amount of time to obtain the least model. In contrast, SizeOpt is much less effective in deleting clauses, and requires the invocation of the least model solver whose run-time dominates the entire process.

5.6 Partial Instantiation Trees: IncrOptFact vs SizeOpt

Thus far, we have only compared IncrOptFact with SizeOpt in those situations where both algorithms are required to process the same number of clauses. But recall that our incremental algorithms are designed for a slightly different purpose: to expand partial instantiation trees efficiently. As described in Section 2.1, if program P in a node N gives rise to conflict-set unifiers $\theta_1, \ldots, \theta_m$, then N has m child nodes, each corresponding to $P \cup P\theta_j$. Thus, as shown in Figure 2, the acid test of the effectiveness of our incremental algorithms is between the time taken for our incremental algorithms to process the clauses in $P\theta_j$ and the time taken for SizeOpt to process all the clauses in $P \cup P\theta_j$. Given the results of the previous series of experiments, we expect IncrOptFact to outperform SizeOpt even more in the expansion of partial instantiation trees. This conjecture is confirmed by the following experiment that fully expands the instantiation tree of the program discussed in Section 2.1.

By applying the heuristics of avoiding redundant node expansion discussed in Section 4.3, our algorithm only needs to process 5 nodes (i.e., the root node, and Nodes 1, 2, 4 and 5), as compared with 11 that would be needed otherwise (cf: Figure 1). This demonstrates the usefulness of the heuristics. The following table compares IncrOptFact with SizeOpt for the expansion of 5 nodes only. In other words, the total run-time taken by SizeOpt to expand 11 nodes would be even higher than the time recorded below. Each entry in the table below gives two run-times: i) the time taken to process the clauses in $P\theta_j$ by IncrOptFact, or in $P \cup P\theta_j$ by SizeOpt; and ii) the time taken to find the least model.

	IncrOptFact	SizeOpt
Node 1 (ms)	0.67/5.47	0.33/45.88
Node 2 (ms)	0.02/5.57	0.34/45.86
Node 3 (ms)	0.02/5.57	0.34/53.95
Node 4 (ms)	0.02/5.49	0.34/49.19
Node 5 (ms)	0.02/5.57	0.34/52.88
total (processing time/model solving time)	0.75/27.67	1.69/247.76
total (processing time + model solving time)	28.42	249.45

As expected, the processing time of IncrOptFact for the first node is relatively long (i.e. 0.67ms), whereas the processing times for subsequent nodes are much shorter (i.e. 0.02ms). This reflects the benefit of being incremental. At the end, the total processing time of IncrOptFact is 0.75ms, less than 50% of that of SizeOpt. Furthermore, as shown in previous experiments, IncrOptFact requires much less time in finding least models. Thus, the conclusion is very obvious and convincing: the time taken to expand the 5 nodes by using IncrOptFact is merely over 10% of the time taken by using SizeOpt.

6 Conclusions

The objective of this paper is to study how to optimize the expansion of partial instantiation trees for computing minimal and least models. Towards this goal, we have developed Algorithm Incr which is formally proved to be incremental. We have further optimized Incr to delete clauses in self-sustaining cycles, to partially order clauses to be inserted, and to avoid expanding redundant nodes. Those optimizations lead to several algorithms, among which experimental results indicate that IncrOptFact gives the best performance. More importantly, when compared with the original algorithm SizeOpt, IncrOptFact can give very significant improvement in run-time efficiency.

In ongoing work, we investigate the optimal order to expand nodes in partial instantiation trees, in terms of both space and time efficiency. In situations where it is not desirable or too costly to generate an entire partial instantiation tree, we will study how to generate portions of the tree selectively.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv and J. Ullman. (1986) *Magic Sets and Other Strange Ways to Implement Logic Programs*, Proc. ACM-PODS, pp 1–15.

[2] F. Bancilhon and R. Ramakrishnan. (1986) *An Amateur's Introduction to Recursive Query Processing Strategies*, Proc. ACM-SIGMOD, pp 16–52.

[3] C. Bell, A. Nerode, R. Ng and V.S. Subrahmanian. (1992) *Implementing Deductive Databases by Linear Programming,* Proc. ACM-PODS, pp 283–291.

[4] C. Bell, A. Nerode, R. Ng and V.S. Subrahmanian. (1992) *Mixed Integer Programming Methods for Computing Nonmonotonic Deductive Databases,* to appear in: Journal of ACM.

[5] J. Blakeley, N. Coburn and P. Larson. (1989) *Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates*, ACM TODS, 14, 3, pp 369–400.

[6] J. Blakeley, P. Larson and F. Tompa. (1986) *Efficiently Updating Materialized Views*, Proc. ACM-SIGMOD, pp 61–71.

[7] R. Boyer and J. Moore. (1972) *The Sharing of Structure in Theorem-proving Programs*, Machine Intelligence, 7, pp 101–116.

[8] Buning and Lowen. (1989) *Optimizing Propositional Calculus Formulas with Regard to Questions of Deducibility,* Information and Computation, 80.

[9] S. Ceri and J. Widom. (1991) *Deriving Production Rules for Incremental View Maintenance*, Proc. VLDB, pp 577–589.

[10] V. Chandru and J. Hooker. (1991) *Extended Horn Sets in Propositional Logic*, Journal of the ACM, 38, 1, pp 205–221.

[11] G. Dong and R. Topor. (1992) *Incremental Evaluation of Datalog Queries*, Proc. ICDT.

[12] M. Gelfond and V. Lifschitz. (1988) *The Stable Model Semantics for Logic Programming*, in: Proc. 5th International Conference and Symposium on Logic Programming, ed R. A. Kowalski and K. A. Bowen, pp 1070–1080.

[13] A. Gupta, I. Mumick and V.S. Subrahmanian. (1993) *Maintaining Views Incrementally*, Proc. ACM-SIGMOD, pp 157–166.

[14] J. Harrison and S. Dietrich. (1992) *Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach*, Workshop of JICSLP.

[15] R. E. Jeroslow. (1988) *Computation-Oriented Reductions of Predicate to Propositional Logic*, Decision Support Systems, 4, pps 183–187.

[16] V.Kagan, A. Nerode and V.S. Subrahmanian (1993) *Computing Definite Logic Programs by Partial Instantiation*, to appear in: Annals of Pure and Applied Logic.

[17] V.Kagan, A. Nerode and V.S. Subrahmanian (1994) *Computing Minimal Models by Partial Instantiation*, draft manuscript, submitted to a technical journal for publication.

[18] J. Lobo, J. Minker and A. Rajasekar. (1992) *Foundations of Disjunctive Logic Programming*, MIT Press.

[19] A. Martelli and U. Montanari. (1982) *An Efficient Unification Algorithm*, ACM Trans. on Programming Languages and Systems, 4, 2, pp 258–282.

[20] A. Nerode, R. Ng and V.S. Subrahmanian. (1992) *Computing Circumscription by Linear Programming*, to appear in: Information and Computation.

[21] O. Shumeli and A. Itai. (1984) *Maintenance of Views*, Sigmod Record, 14, 2, pp 240–255.

[22] A. van Gelder, K. Ross and J. Schlipf. (1988) *Unfounded Sets and Well-founded Semantics for General Logic Programs*, in Proc. ACM-PODS, pp 221-230.

[23] O. Wolfson, H. Dewan, S. Stolfo and Y. Yemini. (1991) *Incremental Evaluation of Rules and its Relationship to Parallelism*, Proc. ACM-SIGMOD, pp 78–87.

[24] C. Zaniolo. (1988) *Design and Implementation of a Logic-based Language for Data-Intensive Applications*, Proc. of the International Conference on Logic Programming (eds. K. Bowen and R. Kowalski), pps 1666-1687, MIT Press.