OPTIMIZATIONS FOR MODEL COMPUTATION BASED ON PARTIAL INSTANTIATION

By

Xiaomei Tian

B. Sc. (Computer Science) Tsinghua University
M. Sc. (Computer Science) Tsinghua University

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in
THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
August 1994
© Xiaomei Tian, 1994
In presenting this thesis in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science
The University of British Columbia
2366 Main Mall
Vancouver, Canada
V6T 1Z4

Date: Sept. 29, 1994
Abstract

Various methods have been presented for efficiently evaluating deductive databases and logic programs. 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. However, the “groundness” requirement is a huge drawback because the ground version of a logic program can be very large when compared to the original logic program. A novel approach, called partial instantiation, has been developed recently which, when integrated with mixed integer programming methods, can handle non-ground logic programs. The goal of this thesis is to explore how this integrated framework based on partial instantiation can be optimized. In particular, we have developed an incremental algorithm that minimizes repetitive computations for reducing the size of a program. We have also developed several optimization techniques to further enhance the efficiency of our incremental algorithm, to further reduce the size of a logic program, and to avoid redundant node expansion in partial instantiation tree. Experimental results have shown that our algorithm and optimization techniques can bring about significant improvement in run-time performance. Last but not least, we have implemented the integrated framework of partial instantiation under UNIX environment.
Table of Contents

Abstract

List of Figures

Acknowledgments

1 Introduction

1.1 Deductive Databases .................................. 1
1.2 Query Evaluation and Optimization ......................... 2
1.3 Partial Instantiation and Thesis Contributions .......... 4

2 Related Works

2.1 Query Evaluation .................................. 7
  2.1.1 Naive Method .................................. 8
  2.1.2 Semi-Naive Method ............................... 8
  2.1.3 Backward Chaining .............................. 8
  2.1.4 Linear Programming ............................ 9
2.2 Query Optimization ................................ 9
  2.2.1 Magic Sets .................................. 10
  2.2.2 Counting Methods ............................. 10
  2.2.3 Partial Instantiation ........................... 11
2.3 Incremental Algorithms .............................. 11
5.2 IncrOptFact vs IncrOptOrder vs IncrOptArb ....................................... 51
5.3 Same Number of Disjunctive Clauses: IncrOptFact vs SizeOpt ........... 52
5.4 Same Number of Definite Clauses: IncrOptFact vs. SizeOpt .......... 54
5.5 Partial Instantiation Trees: IncrOptFact vs. SizeOpt ................. 54
5.6 Partial Instantiation Trees: With and Without Applying Lemma 7 .... 56

6 Implementation Details of Partial Instantiation .................................. 58
   6.1 General Picture ............................................................................. 58
   6.2 Data Structures ........................................................................... 59
       6.2.1 Term and Term Table ............................................................ 60
       6.2.2 Atom and Atom Table ............................................................. 62
       6.2.3 Clause and Clause Table ......................................................... 62
       6.2.4 Tree ..................................................................................... 63
   6.3 Generating New Clauses .................................................................. 64
   6.4 Least/Minimal Model Solver ........................................................... 65
   6.5 Unification ..................................................................................... 66
       6.5.1 Algorithm UNIFY: an Efficient Unification Algorithm ............ 67
       6.5.2 Implementation of Algorithm UNIFY ....................................... 69
   6.6 Cutting Unifiers ............................................................................ 70

7 Conclusions ............................................................................................. 71
   7.1 Thesis Summary .............................................................................. 71
   7.2 Future Work ................................................................................... 72
       7.2.1 Cutting Redundant Nodes ......................................................... 72
       7.2.2 Order for Node Expansion ....................................................... 72
7.2.3 Tree Maintenance ............................................ 73
7.2.4 Partial Tree ................................................... 73

Bibliography ......................................................... 74
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Partial Instantiation Tree</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Incremental Maintenance</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>DC-Graph $G_1$</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Applying Algorithm Incr to Add Clauses 1, 2, 3 and 4</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Applying Algorithm Incr to Add Clauses 4, 3, 2 and 1</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>Applying Algorithm Incr to Add Clauses 5 and 6</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>Cutting Redundant Branches</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>Reduced Partial Instantiation Tree</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>The Structure of Term $f(a, g(X, Y))$</td>
<td>61</td>
</tr>
</tbody>
</table>
Acknowledgments

I would like to express my sincere thanks to my supervisor Dr. Raymond Ng for many things, especially for suggesting the thesis topic, for fruitful discussions and most helpful guidance, and for being a continuous source of advice and encouragement throughout my research work.

Thanks to the Department of Computer Science at the University of British Columbia for providing the research environment. Thanks to Dr. Ng and the department for providing financial support in the form of teaching and research assistantships during my graduate studies.

Thanks to Dr. Craig Boutilier for being the second reader of this thesis. His helpful comments are highly appreciated.
Chapter 1

Introduction

1.1 Deductive Databases

Over the last two decades there has been a lot of research done in the area of deductive database systems [1]. Deductive databases extend conventional relational databases with deductive power by adding clauses into databases from which new facts can be deduced. This extension is based on a very sound theoretical foundation – namely, first order logic. A deductive database can also be considered as a logic program with no function symbols.

Knowledge of the basic theory of logic programming and deductive databases can be found in [2] [3]. Here we give the basic concepts needed throughout.

**Definition.** A Horn clause is a clause of the form

\[ A \leftarrow B_1 \land B_2 \land \ldots \land B_m, m \geq 0, \]

where \( A \) and \( B_1, B_2, \ldots, B_m \) are atoms. \( A \) is called the head of the clause. \( B_1, B_2, \ldots, B_m \) is a conjunction of atoms and is called the body of the clause.

**Definition.** A definite database is a finite set of Horn clauses.

For a given deductive database there are many models. The nice property of a definite database is that among all these models, there is a unique least model \( M_p \), i.e., if \( I \) is a model of the same program, then \( I \succeq M_p \). This least model is the one we choose as the model of the definite database.

The expressive power of Horn clauses is limiting, as Horn clauses are incapable of
supporting negations and disjunctions. A lot of efforts have been done to extend the expressive power of Horn logic [2][4]. Two most common extensions are:

i) adding negations to the right side of the clause and,

ii) allowing more than one atom in the left side of the clause.

**Definition** A *disjunctive clause* is a clause of the form

\[ A_1 \lor A_2 \lor \ldots \lor A_n \leftarrow B_1 \land B_2 \land \ldots \land B_m, n \geq 1, m \geq 0, \]

where \( A_i \) is a disjunction of atoms and \( B_i \) is a conjunction of atoms.

**Definition.** A *disjunctive database* is a finite set of disjunctive clauses.

Unlike Horn databases, a disjunctive databases may not have a unique least model. Instead a disjunctive database may very often have multiple minimal models. \( M \) is a minimal model of a disjunctive database \( P \) iff

i) \( M \) is a model of \( P \) and,

ii) there is no model \( I \) such that \( I \subseteq M \).

As far as computation of minimal models is concerned, an atom with negation in the body of a clause can be moved to become a positive atom in the head, hereafter we only consider clauses with disjunctive heads, but no negations in the bodies.

### 1.2 Query Evaluation and Optimization

Query evaluation and optimization is a crucial area of deductive database systems. Its aim is to characterize complete answers in terms of minimal Herbrand models. Various strategies have been presented in the literature. [5] gives a complete survey on the various strategies. Algorithms, such as magic sets and counting methods, have proven to be very successful for definite and stratified deductive databases. During the past few years, several new semantics for disjunctive programs and programs with negations, such as
minimal models, stable models and well-founded models, have been proposed and widely studied [6] [7] [8]. 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 [9] [10] [11].

Compared with other methods, mixed integer programming methods have many advantages in addition to effectively supporting disjunctive semantics. By translating a symbolic logic problem to a linear programming problem, it takes advantage of all the theorems, algorithms, and software packages that have been developed in the Operations Research community. More importantly, linear programming methods compute minimal models at compile-time, thus improving the run-time performance of deductive databases. Last but not least, linear programming methods are incremental. Adding a new clause to the deductive database amounts to adding a new set of constraints to the associated linear program.

However, like other methods that use linear or integer programming methods for logic deduction [12] [13], the paradigm proposed in [9] [10] [11] is in effect "propositional", and can only deal with the ground versions of deductive databases. The "groundedness" requirement is a huge drawback because the ground version of a logic program can be very large when compared to the original logic program. To solve this problem, [14] and [15] have proposed 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.
Chapter 1. Introduction

1.3 Partial Instantiation and Thesis Contributions

While we will discuss partial instantiation in greater details in Chapter 3, the general strategy of partial instantiation is to alternate iteratively between two phases:

\[ \text{evaluate(propositional program)} \rightarrow \text{partial instantiation} \rightarrow \text{evaluate} \rightarrow ... \]

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 \( \theta \) (there can be multiple), clauses in \( P \) are instantiated with \( \theta \) 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.

The goal of this thesis is to explore how the integrated framework of partial instantiation can be optimized. In particular, our work concentrates on the following two aspects:

(i) Optimize the run-time performance of the evaluation phase, and

(ii) Avoid redundant computations in the partial instantiation phase.

In the evaluation phase, two steps are needed (see [9] [10] [11]). The first step, called \( \text{sizeopt}(P) \), is to reduce the size of the program \( P \); and the second step is to actually find the models by the mixed integer programming method. The first step \( \text{sizeopt}(P) \) is highly beneficial to the subsequent step since the size of program \( P \) is reduced. Suppose
Chapter 1. Introduction

\( \theta_1, \theta_2, \ldots, \theta_n \) are all the conflict set unifiers for \( P \), then in partial instantiation, \( \text{sizeopt}(P \cup P\theta_1), \text{sizeopt}(P \cup P\theta_2), \ldots, \text{sizeopt}(P \cup P\theta_n) \) will be carried out eventually. This leads us to the idea of computing \( \text{sizeopt}(P \cup P\theta_i), 1 \leq i \leq n \), incrementally. That is, we try to optimize the evaluation phase by reusing \( \text{sizeopt}(P) \) to compute \( \text{sizeopt}(P \cup P\theta_i) \).

With regard to the partial instantiation phase, we have noticed that a lot of computations involved in this phase are actually redundant. The main reason for this is that some unifiers in the conflict set only lead to redundant information. In order to optimize the partial instantiation phase, we try to cut off those useless unifiers.

More specifically, the principle contributions of this thesis are:

- We have developed an algorithm, called Incr, which has been formally proved to be incremental with regard to operation \( \text{sizeopt} \).

- We have developed several optimizations which may further reduce the size of a logic program and save time in computing minimal models. For definite programs, it directly gives the solution without computing minimal models.

- We have developed three rules which may reduce the size of the set of conflict unifiers, and thus avoiding redundant nodes expansions in the integrated framework.

- We have implemented these algorithms and optimizations. Experimental evidence have shown that these methods can lead to significant improvement in run-time efficiency.

- We have implement the whole integrated framework, i.e., given a program, the system generates the partial instantiation tree and returns all the facts. Though only handling definite databases at the current stage, it is easy to be extended to disjunctive databases.
The outline of the thesis is as follows. Chapter 2 reviews the related works. Chapter 3 presents the incremental algorithm Incr and proves that it is indeed incremental. Chapter 4 develops several optimizations to further improve the performance of Incr and partial instantiation. Chapter 5 gives implementation details and presents experimental results showing the effectiveness of the algorithms and optimizations. Chapter 6 gives the implementation details of the whole framework of partial instantiation. At last Chapter 7 concludes this thesis by providing a summary of contributions and an outline of future works.
Chapter 2

Related Works

As shown in the first chapter, a major goal of this thesis is the development of an incremental algorithm which optimizes partial instantiation. In this chapter we will review the related works. Specifically, we will first discuss different methods for query evaluation and optimization, and then discuss incremental algorithms for database maintenance.

2.1 Query Evaluation

Evaluating a logic program is to generate the actual set of tuples which satisfy a given user query for the given clauses. Generally there are two evaluation paradigms:

1. bottom-up: starting from the existing facts, inferring new facts, and proceeding towards the goals.

2. top-down: starting from the goals, verifying the premises which are needed in order for the goals to hold, and stop when the premises can be supported by facts.

Top-down paradigm is particularly useful in cases where a goal $G$ is specified, and we are only interested in those facts which are instances in $G$. On the other hand, bottom-up paradigm is very useful in cases where multiple queries are to be answered.

Following are some popular evaluation methods presented in literature. Naive, Semi-naive and Linear Programming methods are bottom-up approaches, while the Backward Chaining method is a top-down approach.
Chapter 2. Related Works

2.1.1 Naive Method

Naive method starts with the facts of a deductive database and then evaluates each clause as follows. When a body of a clause is proven to be true, then the head of the clause can be inferred to be true. When all clauses have been evaluated, we can repeat this process and perform deduction with the clauses using the original facts and derived facts. The process is continued until no new fact is generated.

Naive evaluation is the most widely described method in the literature. It has been presented in a number of papers under different forms [16] [17] [18] [19] [20], etc.

2.1.2 Semi-Naive Method

Semi-naive method uses the same approach as Naive, but tries to eliminate redundancy in the evaluation of tuples at different iterations. It evaluates the differential of a relation instead of the entire relation at each step. The idea of semi-naive underlies many papers. A complete description of the method is given in [21] and [22].

Naive and Semi-Naive methods are quite simple, but they compute a lot of useless results because they do not know what query they are evaluating. On the contrary, the backward Chaining method below only generates facts that may be relevant to the goal.

2.1.3 Backward Chaining

Backward Chaining starts from a goal clause and works towards the facts of a deductive database. The initial goal is unified with the left-hand side of some clause, and generates subgoals corresponding to the right-hand side atoms of that clause. This process is continued until the subgoals can be supported by the facts of the database. In this case only facts that are related to the goal are involved in the computation. Many of the facts
which are not useful for producing the result are disregarded automatically. Prolog is a form of backward chaining.

Backward chaining method is often more efficient than Naive and Semi-Naive methods because it takes advantage of the existence of bound arguments in the goal predicate. However, it performs all deductions at run-time, and thus not suitable for applications where run-time performance is critical. The following method solves this problem.

2.1.4 Linear Programming

The idea and implementation of linear programming method can be found in [9] [10] [11]. In Linear Programming method, deductive databases are translated into sets of linear constraints. A solution to the derived linear program corresponds to a model of a deductive database.

As mentioned in the previous chapter, compared with the other methods, linear programming method has many advantages, such as supporting semantics of minimal models, stable models and well-founded models, performing most deductions at compile-time and thus improving the run-time efficiency, etc. Its main problem has been the infamous "grounding" problem. The optimization method of partial instantiation solves this problem by providing a "instantiate-by-need" theory. The optimization of the partial instantiation is the goal of this thesis.

2.2 Query Optimization

Query optimization transforms a program into another program which is written in the same formalism, but which yields a more efficient computation when one applies an evaluation method to it. These methods are different from the pure evaluation methods
which propose effective evaluation strategies.

2.2.1 Magic Sets

The idea of the Magic Sets optimization is to cut down on the number of potentially relevant facts by simulating the sideways passing of bindings. In order to simulate the binding passing strategy of top-down methods, the Magic Sets introduce constraints into the program, by means of additional subgoals added to the right hand side of the original clauses, and additional clauses defining these goals added to the program. The additional clauses constrain the program variables to satisfy other predicates (called "magic" predicates). Thus, during bottom-up computation, the variables assume only some values instead of all possible ones. In most cases, this makes the new program more efficient. The idea of the magic set strategy was presented in [23] and the precise algorithm is described in [24].

2.2.2 Counting Methods

The Counting method is a rewriting method based on the knowledge of the goal bindings; the method includes the computation of the Magic Set, but each element of the Magic Set is complemented by additional information expressing its "distance" from the goal element. Instead of using the entire magic set, we only use the tuples of the correct level, thus minimizing the set of relevant tuples. The idea of counting was presented in [23]. Variations and extensions can be found in [25].

Magic sets and counting methods are very successful for definite and stratified databases. However, they do not support the newly developed semantics such as minimal models
and stable models. The following method, partial instantiation, is aimed at optimizing linear programming evaluation method which supports these new semantics as well.

2.2.3 Partial Instantiation

The idea of Partial Instantiation is presented in [14] [15]. As we already know, query processing in a ground deductive database corresponds precisely to a linear programming problem. However, the "groundness" requirement is a huge drawback to using linear programming techniques for logic computations because the ground version of a logic program can be very large when compared to the original logic program. Partial Instantiation is based on the theory of "instantiate-by-need", that is, performing instantiation (not necessarily ground instantiations) only when needed.

The goal of this thesis is to develop optimization methods for partial instantiation. A detailed discussion about partial instantiation will be given in Chapter 3.

2.3 Incremental Algorithms

Excellent work has been done on incremental maintenance for relational, active and deductive databases [26] [27] [28] [29] [30] [31] [32] [33]. In the following we will first briefly introduce these works and then point out the fundamental differences between these works and the work reported in this thesis.

[30] has proposed two algorithms: a counting algorithm for nonrecursive views, and a delete and re-derive algorithm (DRed) for recursive views. The counting algorithm tracks the number of alternative derivations (counts) for each derived tuple in a view. The DRed algorithm computes an over-estimate of tuples that need to be deleted, and then re-derives some of them.
Chapter 2. Related Works

[32] has proposed a mechanism which dynamically maintains a class of views. It quickly updates the view in response to database changes by minimizing the amount of work caused by updates.

[27] concerns with how to update views efficiently. It first detects and removes those database updates that cannot possibly affect the view. For the remaining database updates, a differential algorithm is given to re-evaluate the view expression.

[33] deals with incremental evaluation of clauses in deductive databases. Its major concern is when changes to the facts of deductive databases take place, how to efficiently incorporate these changes to the inference procedures. An algorithm called INCR.UPDATE is presented to maintain the inference database incrementally.

Other related works include [31] which deals with recursive views and [29] whose concern is right-linear chains, etc.

All the proposals listed above 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 in this thesis focus on handling clauses inserted or deleted. Second, the operation under consideration in this thesis is not logic deduction, i.e. deducing heads from the bodies of clauses. Rather, as will be discussed in greater detail in Chapter 3, the operation sizeopt takes a set of clauses $P$ as input, and returns a subset $P' \subseteq P$ by deleting clauses that will not be useful in subsequent model computations.

In the following chapter we will discuss in detail the incremental algorithm we have developed, and then give a formal proof of its correctness.
Chapter 3

An Incremental Algorithm

In this chapter we will first formalize the partial instantiation and the optimization algorithm \textit{sizeopt} for evaluation in greater details. We will then present the most important algorithm of this thesis – Algorithm \textit{Incr}, and will prove that \textit{Incr} is indeed incremental with regard to \textit{sizeopt}.

3.1 Preliminaries

3.1.1 Basic Definitions

\textbf{Definition (Unifier [34])} Two atoms (or expressions) A and B are unifiable if there is a substitution $\theta$ such that $A\theta = B\theta$. The substitution $\theta$ is called a unifier for A and B. \hfill $\square$

\textbf{Definition (Most General Unifier [34])} A unifier for two atoms A and B is called a most general unifier (mgu) if for each unifier $\eta$ for A and B there exists a substitution $\gamma$ such that $\eta = \theta\gamma$. \hfill $\square$

\textbf{Definition (Disagreement Set [14])} Suppose T and F are two sets of atoms. The disagreement set of T, F, denoted by DIS(T,F), is the set \{ $\theta$ | there exist atoms $A_1 \in T$ and $A_2 \in F$ such that $A_1$ and $A_2$ are unifiable via mgu $\theta$ \}. \hfill $\square$
3.1.2 Partial Instantiation

Partial instantiation is aimed at solving the "groundness" problem of linear programming method. As described in [14] [15], partial instantiation computes minimal models of logic programs by expanding and processing nodes of partial instantiation trees; a logic program (not necessarily ground) is evaluated as a propositional program at each tree node.

**Definition (Partial Instantiation Tree [15])** Given a logic program \( P \) (definite or disjunctive), we define the normal partial instantiation tree, NPIT\((P)\), associated with \( P \) as follows:

1. Each node, \( N \), in NPIT\((P)\) is labeled with a pair \((P_N, S_N)\) and:
   
   (a) \( P_N \) is a logic program (definite or disjunctive).

   (b) \( S_N \) is a set of pairs of the form \((T_N, F_N)\) where:

   i. For each \( T_{N_j} \), there is a minimal model, \( M_j \), of \( P_N \), such that \( T_{N_j} = \{ A | A \in M_j \} \). (Note that \( P_N \) is treated propositionally here and the minimal models of \( P_N \) can be computed using various methods such as the integer linear programming algorithm described in [9]).

   ii. \( F_{N_j} = \{ A | A \text{ is an atom occurring in } P_N \text{ such that } A \text{ is not } \in T_{N_j} \} \).

2. The root of NPIT\((P)\) is labeled with the pair \((P, S)\) where \( P \) is the original logic program. All subsumed clauses in \( P \) are deleted.

3. If node \( N \) is labeled with the pair \((P_N, S_N)\), \( N \) is a leaf node and has no children if either of the following conditions stands:

   (a) for all \( T_{N_i}, F_{N_i} \in S_N \), the disagreement sets of \( T_{N_i}, F_{N_i} \) are empty.
Chapter 3. An Incremental Algorithm

(b) N is an exact copy of some other node on the path from the root of the tree to N.

4. Otherwise, there exists a pair \((T_{N_j}, F_{N_j}) \in S_N\) such that the disagreement sets of \(T_{N_j}, F_{N_j}\) are not empty. For each such \(j\) and each substitution \(\theta\), node \(N\) has a child, \(N^j\) labeled with the pair \((P^j_{\theta}, S^j_{\theta})\) defined below:

(a) \(P^j_{\theta}\) is constructed as follows: \(P^j_{\theta} = P_N \cup \{C\theta|C \in P_N\}\).

(b) \(P^j_{\theta}\) uniquely determines \(S^j_{\theta}\).

The following example shows how partial instantiation works.

Example 1 Let \(P\) be the program consisting of clauses:

\[
\begin{align*}
p(X_1,Y_1) & \leftarrow q(X_1,Y_1) \\
q(a,Y_2) & \leftarrow \\
q(X_2,b) & \leftarrow
\end{align*}
\]

At the root node, \(P\) is considered as a propositional program

\[
\begin{align*}
A & \leftarrow B \\
C & \leftarrow \\
D & \leftarrow
\end{align*}
\]

where \(A, B, C, D\) denote \(p(X_1,Y_1), q(X_1,Y_1), q(a,Y_2), q(X_2,b)\) respectively. For this propositional program, the set 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) \(\theta_1 = \{X_1 = a, Y_1 = Y_2\}\)

(b) \(\theta_2 = \{X_1 = X_2, Y_1 = b\}\)

Now for each conflict-set unifier \(\theta_i\), a child node is created which is responsible for the processing of the instantiated program \(P \cup P\theta_i\). Thus, for our example, the root node
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 \( \theta_1, \theta_2 \). 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, or if it is an exact copy of a previous node. For our example, the partial instantiation tree is finite and has 11 nodes in total, as shown in Figure 3.1. \( \Box \)

3.1.3 Algorithm SizeOpt

As far as computing minimal models is concerned, some clauses in a logic program are not useful and thus can be deleted. Suppose there is an atom \( A \) that does not appear in the head of any clause of a program \( P \), it is easy to see that \( A \) cannot be in any minimal model of \( P \). Thus, those clauses with \( A \) in their bodies are never useful, and can therefore be thrown away. The following algorithm, first developed in [9], intends to reduce the size of a given program by deleting clauses whose bodies cannot possibly be satisfied.

**Algorithm SizeOpt** [9]

**Input:** \( P \), a ground disjunctive program, and \( S_0 \), the set of atoms that do not appear in the head of any clause in \( P \).

**Output:** \( Q \), the set of retained clauses, and \( Q_d \), the set of deleted clauses.
Chapter 3. An Incremental Algorithm

1. Initialize $Q$ to $P$, $Q_0$ to $\emptyset$ and $i$ to 0.

2. Set $R$ to $\emptyset$ ($R$ is used to collect the heads of the deleted clauses at each iteration).

3. For each clause $Cl \equiv A_1 \lor ... \lor A_m \leftarrow B_1 \land ... \land B_n$ 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 $sizeopt(P) = (Q, Q_d)$ 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 2 Let $P$ be the following program:

\begin{align*}
A & \leftarrow B \land C & \quad (3.1) \\
B \lor D & \leftarrow A \land E & \quad (3.2) \\
B & \leftarrow E \land F & \quad (3.3) \\
D & \leftarrow A & \quad (3.4)
\end{align*}

Initially, $S_0$ is the set $\{C, E, F\}$. 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 3 Let $P'$ be the program obtained by adding the following two clauses to $P$ introduced in the previous example:

\begin{align*}
C \lor G & \leftarrow & \quad (3.5) \\
E & \leftarrow C & \quad (3.6)
\end{align*}
Chapter 3. An Incremental Algorithm

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. $P_1 \subseteq P_2$ doesn't necessarily mean $Q_{d,1} \subseteq Q_{d,2}$ where $sizeopt(P_1) = (\neg, Q_{d,1})$ and $sizeopt(P_2) = (\neg, Q_{d,2})$. It is also easy to see that Algorithm SizeOpt is not anti-monotonic either (i.e. $P_1 \subseteq P_2$ doesn't imply $Q_{d,2} \subseteq Q_{d,1}$). The following lemma, proved in [9], shows that Algorithm SizeOpt preserves minimal models.

**Lemma 1** [9] Let $P$ be a disjunctive deductive database such that $Sizeopt(P) = (Q, Q_d)$. $M$ is a minimal model of $P$ iff $M$ is a minimal model of $Q$.

### 3.2 Incremental Algorithm

As shown in [9] [10] [11], the operation $sizeopt(P)$ is highly beneficial to the subsequent operation of finding the models for $P$ since the size of program $P$ is reduced. Suppose $P$ is the program considered in a node $N$ of a partial instantiation tree, and $\theta_1, ..., \theta_m$ are all the conflict-set unifiers. Node $N$ has $m$ children, the $j$-th of which corresponds to 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 3.2:

As shown in Example 2.3, $sizeopt(P)$ is not a monotonic operation, i.e., compared
with SizeOpt(P), SizeOpt(P ∪ Pθ) may either delete more clauses or less clauses. This complicates our incremental algorithm.

3.2.1 Graphs for Maintaining Deleted Clauses

Recall from Chapter 3.1.2 that sizeopt(P) produces the pair ⟨Q, Qd⟩, where Q consists of clauses retained in P and Qd consists of clauses deleted from P. To facilitate incremental processing, Algorithm Incr uses a directed graph G, called DC-Graph, to organize the deleted clauses in Qd. 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 to A, then the arc is labeled by a clause Cl ∈ Qd such that A appears in the head of Cl and B occurs in the body of Cl.

In other words, every arc(clause) in G is from Qd, and every node(atom) in G only appears in the bodies of Q or does not appear in Q at all. 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. It points to the "level 0" iteration in SizeOpt. Arcs originated from the root node are not labeled, because those arcs do not correspond to any clause in Qd.
Intuitively, DC-Graph can be explained this way: if there is an arc $B \xrightarrow{C_l} A$ in DC-Graph, then $C_l$ is the clause deleted from the original program, $B$ is the reason why $C_l$ should be deleted, and $A$ is the new atoms that might cause more clauses to be deleted in next iteration.

**Example 4** Consider the program $P$ discussed in Example 2. $Q_d$ consists of all 4 clauses in $P$. Figure 3.3 shows the DC-Graph $G_1$ corresponding to $Q_d$. For convenience, arcs are labeled by the clause numbers used in Example 2.1. 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.

![Figure 3.3: DC-Graph $G_1$](image)

This example only illustrates how DC-Graph $G_1$ looks like. We will show in Example 5 how $G_1$ can be constructed, after Algorithm Incr has been presented. However, before we can present the algorithm, we need the following concept.
3.2.2 Self-sustaining Cycles

**Definition (Self-sustaining cycle).** Let \( A_1 \overset{C_l_1}{\rightarrow} A_2 \overset{C_l_2}{\rightarrow} ... \overset{C_l_i}{\rightarrow} A_i \overset{C_l_{i-1}}{\rightarrow} A_n \overset{C_l_{i-1}}{\rightarrow} A_1 \) be a cycle in DC-Graph \( G \), where \( A \overset{C_l}{\rightarrow} B \) denotes an arc from \( A \) to \( B \) with label \( C_l \). If there does not exist any arc from outside the cycle to some \( A_i \) with label \( C_l_{i-1} \) (i.e. there is no such \( B \) that \( B \) is outside \( \{A_1, ..., A_n\} \) and \( B \overset{C_l_{i-1}}{\rightarrow} A_i \)), then the cycle is called self-sustaining. □

As shown in the above example, \( G_1 \) contains the cycle \( A \overset{C_l_2}{\rightarrow} B \overset{C_l_1}{\rightarrow} A \). This cycle is not self-sustaining because of the arc \( C \overset{C_l_1}{\rightarrow} A \) (or the arc \( E \overset{C_l_2}{\rightarrow} B \)). The existence of this arc justifies why Clause 1 should be deleted, and why \( A \) should remain as a node in the graph. On the other hand, if the arcs \( C \overset{C_l_1}{\rightarrow} A \) and \( E \overset{C_l_2}{\rightarrow} B \) were removed, the cycle \( A \overset{C_l_2}{\rightarrow} B \overset{C_l_1}{\rightarrow} A \) became self-sustaining. Then for the sake of achieving the kind of incrementality depicted in Figure 3.2, 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 6 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. Notice that DC-Graph \( G \) is used by Incr to judge whether a new clause \( C_l \) should be deleted or not from the original program.

3.2.3 Algorithm Incr

**Algorithm Incr**

**Input:** \( P = (Q, Q_d) \), the DC-Graph \( G \) corresponding to \( Q_d \), and a clause \( C_l \equiv A_1 \lor ... \lor A_m \leftarrow B_1 \land ... \land B_n \) to be added to \( P \).

**Output:** updated \( Q, Q_d \) and \( G \).
Chapter 3. *An Incremental Algorithm*

1. For each $B_i$ that does not appear in $Q$ and $Q_d$ (i.e. appearing for 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 in $G$,
   
   (a) For each $A_j$ where $1 \leq j \leq m$,
   
   (i) If $A_j$ does not appear in $Q$ and $Q_d$ (i.e., appearing for the first time), 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.

**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. $A \overset{Cl}{{\rightarrow}} B$),
   
   (a) Remove the arc from $G$.

   (b) If there does not exist another arc pointing to $B$ with label $Cl$ (i.e. there isn't a $D$ such that $D \overset{Cl}{{\rightarrow}} B$),

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

\[\square\]
(ii) Call subroutine Remove(B) recursively.

3. For each clause Cl in $Q_d$ such that A appears in the body of Cl, if no atom in the
   body of Cl remains as nodes in G, remove Cl from $Q_d$, and add it to Q. □

Hereafter, we use the notation $\text{incr}((Q, Q_d, G), Cl) = (Q^{\text{out}}, Q_d^{\text{out}}, G^{\text{out}})$ to denote the
input and output of Algorithm Incr where:

- $Q$ — the original set of retained clauses
- $Q_d$ — the original set of deleted clauses
- $G$ — the DC-Graph corresponding to $Q_d$
- Cl — the clause to be inserted
- $Q^{\text{out}}$ — new set of retained clauses
- $Q_d^{\text{out}}$ — new set of deleted clauses
- $G^{\text{out}}$ — 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 5 Apply Algorithm Incr to the 4 clauses in the program P discussed in Example 2. In Figure 3.4, the first DC-Graph (labeled (i)) is graph $G_{r_1}$ where
$\text{incr}((\emptyset, \emptyset, \emptyset), Cl_1) = (\emptyset, \{Cl_1\}, G_{r_1})$. 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 3.4 is DC-Graph $G_{r_2}$ where $\text{incr}((\emptyset, \{Cl_1\}, G_{r_1}), Cl_2) =
(\emptyset, \{Cl_1, Cl_2\}, G_{r_2})$. 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 $G_{r_2}$, the cycle is not self-sustaining. It is also not difficult to verify
that $\text{sizeopt}\{\{Cl_1, Cl_2\}\} = (\emptyset, \{Cl_1, Cl_2\})$. 
Similarly, the third graph in Figure 3.4 is produced by applying Algorithm Incr to add $Cl_3$ to $G_{r_2}$, and the fourth one (called $G_1$ in Example 4) is produced by applying Incr to $Cl_4$ and the third Graph.

![Figure 3.4: Applying Algorithm Incr to Add Clauses 1, 2, 3 and 4](image)

![Figure 3.5: Applying Algorithm Incr to Add Clauses 4, 3, 2 and 1](image)

Figure 3.4 shows how DC-Graphs are constructed when Clauses 1, 2, 3 and 4 are inserted. The graphs in Figure 3.5 show the DC-Graphs obtained by inserting the 4 clauses in the reverse order. As expected, the fourth DC-Graphs in Figure 3.4 and Figure 3.5 are the same. Later we will show that inserting the clauses in different orders give identical result.
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 with being added to the set $Q$. This addition may trigger a series of node removals and the shrinkage of $Q_d$.

**Example 6** Now consider program $P'$, that is by adding Clauses 5 and 6 discussed in Example 3. 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 $\text{incr}(\emptyset, \{Cl_1, ..., Cl_4\}, G_1, Cl_5) = (\{Cl_5\}, \{Cl_1, ..., Cl_4\}, G_{r_5})$, where $G_{r_5}$ is the first DC-Graph shown in Figure 3.6. Before we proceed, note that it is not difficult to verify that $\text{sizeopt}({Cl_1, ..., Cl_5}) = (\{Cl_5\}, \{Cl_1, ..., Cl_4\})$.

![Figure 3.6: Applying Algorithm Incr to Add Clauses 5 and 6](image-url)
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 3.6 shows the situation at this point.

However, unlike the above situation for Clause 6, this time the cycle between $A$ and $B$ is self-sustaining. Thus, in Step 4 of Algorithm Incr, Subroutine Remove($B$) is called. 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 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 removed 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 terms, we have $\text{incr}((\{Cl_5\}, \{Cl_1, ..., Cl_4\}, G_{r_n}), Cl_6) = (\{Cl_1, Cl_2, Cl_4, Cl_5, Cl_6\}, \{Cl_3\})$, where $G_{r_n}$ is the last DC-Graph shown in Figure 3.6.

As shown in Example 3, we have $\text{sizeopt}(\{Cl_1, ..., Cl_6\}) = (\{Cl_1, Cl_2, Cl_4, Cl_5, Cl_6\}, \{Cl_3\})$, 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 figure3.6, but not as in the third graph.

**Example 7** Thus far, we have not seen a situation in which Step 3 of Subroutine Remove is needed. But given the third graph in Figure 3.6, 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 $Q_d$. 

3.3 **Correctness Proof: Incrementality of Algorithm Incr**

In the remainder of this section, we will 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.

Before we can prove the theorem, we need the following lemmas.

3.3.1 **Supporting Lemmas**

**Lemma 2** Let $P$ be the set $\{CL_1, ..., CL_n\}$. Then:

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

2. Let $incr(...incr((\emptyset, \emptyset, \emptyset), CL_1), ..., CL_n) = (P_n, P_{n,d}, G_n)$. It is the case that $P_n \cup P_{n,d} = P$ and $P_n \cap P_{n,d} = \emptyset$.

**Proof** 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 $A_1 \lor \ldots \lor A_m \leftarrow$, then by
Step 3, $P_1 = \{Cl_1\}$ and $P_{1,d} = \emptyset$. Otherwise, $Cl_1$ is of the form $A_1 \lor \ldots \lor A_m \leftarrow B_1 \land \ldots \land B_u$.

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 $\text{incr}(\ldots \text{incr}(\langle \emptyset, \emptyset, \emptyset\rangle, Cl_1), \ldots, Cl_n) = \langle P_n, P_{n,d}, G_n\rangle$. 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** 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
Chapter 3. An Incremental Algorithm

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 two cases.

Case 1 Subroutine Remove is not called. For any atom A, there are two subcases.

Case 1.1 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 cause 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.

Case 1.2 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 \( P_k \).
Chapter 3. An Incremental Algorithm

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, $B \rightarrow A$ 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.

3.3.2 Rank: Bridge Between Incr and SizeOpt

Definition (Rank). Let A be a node in a DC-Graph G. The rank of A in G, denoted by $\text{rank}(A)$, is defined recursively as follows:

1. If there is an arc from the root to A, $\text{rank}(A) = 0$;
2. Let $B_{1,1},...,B_{1,u_1},...,B_{m,1},...,B_{m,u_m}$ be all the nodes that have arcs pointing to A, such that: a) $\{Cl_1,...,Cl_m\}$ are all the labels of these arcs, and b) for all $1 \leq j \leq m$, $B_{j,1},...,B_{j,u_j}$ are all the nodes that have arcs pointing to A with label $Cl_j$. Then $\text{rank}(A) = 1 + \max_{j=1}^m (\min_{i=1}^{u_j} \text{rank}(B_{j,i}))$. \hfill $\Box$

Example 8 Consider the DC-Graph $G_1$ introduced in figure 3.4. The nodes with $\text{rank} = 0$ are C, E and F. Now consider $\text{rank}(A)$. There are the arcs from C and B pointing to A, both with label $Cl_1$. Thus, $\text{rank}(A) = 1 + \min\{\text{rank}(C), \text{rank}(B)\}$. Since $\text{rank}(C) = 0$, it is obvious that $\text{rank}(A) = 1 + \text{rank}(C) = 1$. Now consider $\text{rank}(B)$ 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 $\text{rank}(A)$, the minimum corresponding to $Cl_2$ is $\text{rank}(E) = 0$. For $Cl_3$, there are the arcs from E and F to B. Thus, the minimum based on $Cl_3$ is $\min\{\text{rank}(E), \text{rank}(F)\} = 0$. Hence, $\text{rank}(B) = 1 + \max\{0,0\} = 1$, where the two zeros correspond to $Cl_2$ and $Cl_3$ respectively. Similarly, it is not difficult to verify that $\text{rank}(D) = 1 + \text{rank}(A) = 2$. Now compare the ranks with the set $S_0, S_1$ and $S_2$ discussed in Example 2.2. The interesting thing here is that for all atoms A, $\text{rank}(A) = k$ 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 3.6. Then $\text{rank}(B)$ depends on $\text{rank}(A)$ which in turn depends on $\text{rank}(B)$. 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 the definition of self-sustaining cycle, for the non self-sustaining cycle
Chapter 3. An Incremental Algorithm

33

\[ A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_{i-1} \rightarrow A_i, \] there must exist atoms \( A_i \) such that there exists arc \( B \rightarrow A_i \) for some atom \( B \notin \{A_1, \ldots, A_n\} \). Thus, in determining \( \text{rank}(A_i) \), for Clause \( Cl_{i-1} \), \( \min\{\text{rank}(B), \text{rank}(A_{i-1})\} \) is always well-defined (cf. the previous example). Thus, there is no cyclic dependency on rank assignments.

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

Proof Prove by induction on \( n \). When \( n = 0, \text{rank}(A) = 0 \) 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 \( Q^{out} \cup Q^{out}_d \). Then when applying Algorithm SizeOpt directly on \( Q^{out} \cup Q^{out}_d \), 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.

Case 1 \( \text{rank}(A) = k \)

By the definition of rank, \( \text{rank}(A) = 1 + \max_{j=1}^m (\min_{v=1}^u \text{rank}(B_{j,v})) \). 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 \( \text{rank}(A) = k = 1 + (\min_{v=1}^u \text{rank}(B_{j,v})) \). More specifically, \( Cl_j \) must be of the form \( \ldots A_{i-1} \rightarrow \ldots \rightarrow A_j \rightarrow \ldots A_{i-1} \). Among these \( u_j \) atoms, let \( i \) be the one so that \( \text{rank}(B_{j,i}) = \min_{v=1}^u \text{rank}(B_{j,v}) \). In other words, \( \text{rank}(B_{j,i}) = k - 1 \).

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 specially, 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 $S_k$.

Case 2 $A \in S_k$.

As shown in Algorithm SizeOpt, there must exist a clause $C_{l_j}$ of the form $\ldots A \leftarrow \ldots B_{j,i}, \ldots$, 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, $\text{rank}(B_{j,i}) = k - 1$. Now among all $B_{j,1}, \ldots, B_{j,w}$ that appear in the body of $C_{l_j}$ and that appear as nodes in the DC-Graph, suppose there exists $B_{j,t}$ such that $\text{rank}(B_{j,t}) < k - 1$. By the induction assumption, $B_{j,t} \in S_w$ where $w < k - 1$. In that case, by Step 3 of Algorithm SizeOpt, the clause $C_{l_j}$ must have been deleted earlier, and should not exist for deletion in the current iteration. This is a contradiction. Thus, it is necessary that $\text{rank}(B_{j,i}) = \underset{v=1}{\overset{w}{\min}} \text{rank}(B_{j,v})$. By applying a similar argument, for every clause $C_{l_w}$ among $C_{l_1}, \ldots, C_{l_m}$ with A in the heads, there exists an $I_w$ for $1 \leq w \leq m$ such that $\text{rank}(B_{w,i_w}) = \underset{v=1}{\overset{w}{\min}} \text{rank}(B_{w,v})$. But since $C_{l_j}$ is the last clause to be deleted, it is necessary that $\text{rank}(B_{j,i}) = \text{rank}(B_{j,i_j}) = \max\{B_{1,i_1}, \ldots, B_{m,i_m}\}$. Hence, it is necessary that $\text{rank}(A) = 1 + \text{rank}(B_{j,i}) = k$. 

3.3.3 Proof of Incrementality

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 $C_{l_1}, \ldots, C_{l_n}$. Let $\text{sizeopt}(P) = \langle Q, Q_d \rangle$, and $\text{incr}(\ldots \text{incr}(\langle \emptyset, \emptyset, \emptyset \rangle, C_{l_1}), \ldots, C_{l_n}) = \langle P_n, P_{n,d}, G_n \rangle$. Then $Q = P_n$ and $Q_d = P_{n,d}$.

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

Chapter 3. An Incremental Algorithm

Case 1 No clause in Q with A in the head.

Then all clauses with A in the head are in Qd, and for some k, A ∈ Sk. By Lemma 4, this is true iff \( \text{rank}(A) = k \). 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 Qd 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 \( \text{rank}(B_j) = k \). 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 \( \text{rank}(B_j) \) 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 case 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
Chapter 3. An Incremental Algorithm

$P_{n,d}$. This completes the proof of the theorem.

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

In this chapter, we have presented the incremental algorithm Incr and formally proved its incrementality. In Incr, we have developed a DC-Graph to organize the deleted clauses and the concept of self-sustaining cycle to achieve the incrementality. In order to prove that Incr is indeed incremental, we have brought about the concept of rank, which connects Incr and SizeOpt. In the next chapter, we will show how algorithm Incr can be optimized.
Chapter 4

Optimizations of the Incremental Algorithm and Partial Instantiation

In the last chapter, we have presented Algorithm Incr and showed that it achieves the kind of incrementality shown in Figure 3.2. In this chapter, we will develop several ways to optimize this algorithm. First we will study how self-sustaining cycles could be retained in DC-Graph. Then we will consider the different orders for inserting new clauses. Finally we will explore the optimization of partial instantiation by cutting down redundant nodes in partial instantiation trees.

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 6, 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 kept 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 \( A_1 \stackrel{C_{l_1}}{\rightarrow} A_2 \stackrel{C_{l_2}}{\rightarrow} \ldots \stackrel{C_{l_i}}{\rightarrow} A_{i+1} \ldots A_n \stackrel{C_{l_r}}{\rightarrow} A_1 \) be a self-sustaining cycle in \( G \). \( M \) is a minimal model of \( Q \cup \{C_{l_1}, \ldots, C_{l_r}\} \) iff \( M \) is a minimal model of \( Q \).
Chapter 4. Optimizations of the Incremental Algorithm and Partial Instantiation

Proof 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 $A_1, ..., A_n$ are nodes in DC-Graph $G$, none of $A_1, ..., A_n$ appears in $Q$. Thus, given any minimal model $M$ of $Q$, none of $A_1, ..., A_n$ contained in $M$. Then it is easy to see that $M$ is a model of $Cl_1, ..., Cl_n$. Hence $M$ is a minimal model of $Q \cup \{Cl_1, ..., Cl_n\}$ iff $M$ is a minimal model of $Q$. □

From the above Lemma, we can see that taking Step 4 out of Algorithm Incr doesn't affect the minimal model. This motivates the following Algorithm.

Algorithm IncrOpt Exactly the same as Algorithm Incr, but without Step 4 of Incr. □

One may ask why we include Step 4, i.e., removing all the sustaining-cycles from DC-Graph, in Algorithm Incr in the first place. Recall that the concept of self-sustaining cycle is brought out to achieve the incrementality. Algorithm Incr can not be formally proved to be incremental without Step 4. However, as shown in Lemma 5, this step is not necessary for computing minimal models. Hereafter, we use the notation $incropt((Q, Q_d, G), Cl) = (Q^out, Q_d^{out}, G^{out})$ for Algorithm IncrOpt in exactly the same way as we use $incr((Q, Q_d, G), Cl) = (Q^out, Q_d^{out}, G^{out})$ 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 $Cl_1, ..., Cl_n$, and let $incropt(... incropt((\emptyset, \emptyset, \emptyset), Cl_1), ..., Cl_n) = (P_n, P_n, G_n)$. $M$ is a minimal model of $P$ iff $M$ is a minimal model of $P_n$. □

As far as supporting minimal model computations is concerned, Algorithm IncrOpt is more preferable than Algorithm Incr for the following reasons:

- 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((Q, Q_d, G), Cl) = (Q^{out}_{opt}, \rightarrow, G^{opt}_{opt}) and incr((Q, Q_d, G), Cl) = (Q^{out}, \rightarrow, \rightarrow), then it is necessary that Q^{out}_{opt} \subseteq Q^{out}. More precisely, IncrOpt keeps all clauses in self-sustaining cycles deleted. Thus, the size of the program Q^{out}_{opt} may be much smaller than Q^{out}. The implication is that finding the minimal models based on Q^{out}_{opt} 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 program P that are definite (i.e. no disjunctive heads). The following lemma shows that for definite programs, Algorithm IncrOpt directly finds their least models.

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

**Proof** 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 Cl_1 is of the form A \leftarrow B_1 \land ... \land 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 \land ... \land 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.
Chapter 4. Optimizations of the Incremental Algorithm and Partial Instantiation

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 $Cl_1,...,Cl_k$. 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 $Cl_1,...,Cl_{k-1}$. By combining the two cases, it is necessary that the least model of $Cl_1,...,Cl_k$ is the same as the least model of $Cl_1,...,Cl_{k-1}$. 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 $A \leftarrow B_1 \land ... \land B_m$. 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 $Cl_1,...,Cl_{k-1}$. Thus, $A$ is in the least model of $Cl_1,...,Cl_k$.

Now consider the subcase when Subroutine Remove is called. A clause Cl may be added to $P_k$ is Step 2b or 3 of Remove. If Cl is added in Step 2b, Cl is of the form $B \leftarrow A \land B_1 \land ... \land B_m$ 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, $B_1,...,B_m$ 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 or 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. \(\square\)
The above lemma 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 9** Consider the definite program

\[
\begin{align*}
A & \leftarrow B, \quad (4.1) \\
B & \leftarrow A, \quad (4.2) \\
C & \leftarrow, \quad (4.3) \\
D & \leftarrow C. \quad (4.4)
\end{align*}
\]

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 clauses 3 and 4 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 \(C_{l_1}, \ldots, C_{l_n}\), and \(\text{incropt}(\ldots \text{incropt}(\emptyset, \emptyset, C_{l_1}), \ldots, C_{l_n}) = (P_n, P_{n,d}, G_n)\), 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 in no. Consider the following program \(P\):

\[
\begin{align*}
A \vee B & \leftarrow, \quad (4.5) \\
A & \leftarrow, \quad (4.6) \\
C & \leftarrow B. \quad (4.7)
\end{align*}
\]

Applying IncrOpt does not cause any change. All the three clauses are retained. 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 \( P \).

### 4.2 Heuristics: Ordering Clauses to Be Inserted

#### 4.2.1 Complexity Analysis

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 \( A_1 \lor \ldots \lor A_m \leftarrow B_1 \land \ldots \land B_l \), the complexity of Algorithm IncrOpt is \( O(ml) \). Otherwise, suppose \( N \) is the number of clauses in the current program, and \( a \) 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 occur at most once – since once \( A \) is removed from the graph, it can never come back. 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. 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 Chapter 5, we will present experimental results evaluating the effectiveness of Algorithm IncrOpt.
The above complexity analysis is rather coarse-grained since \( a \) and \( N \) vary at each iteration. Still it 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.

### 4.2.2 Clauses Arbitrarily Ordered

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 as IncrOptArb.

**Algorithm IncrOptArb** Let \( C_{l_1}, C_{l_2}, \ldots, C_{l_n} \) be the clauses to be inserted. Call IncrOpt \( n \) times to insert \( C_{l_1}, C_{l_2}, \ldots, C_{l_n} \) based on their input order. \( \square \)

### 4.2.3 Clauses Fully Ordered

To the other extreme of arbitrary ordering, another way to insert \( n \) clauses is to really try to minimize the number of times Subroutine Remove will be called.

Subroutine Remove is called when any atom \( A \), previously in DC-Graph, turns out to be in the head of some newly inserted clause \( Cl \) and thus should not stay in the DC-Graph any more. If clause \( Cl \) is inserted first, \( A \) will not be in the DC-Graph at the first place, so no removal will be needed. This leads us to the idea of handling atoms/clauses that cannot be in DC-Graph or more unlikely to be in DC-Graph ahead of other atoms/clauses.

Obviously all the facts of a program must be in the retained clauses and will not appear in DC-Graph. So the facts should be inserted first. Consequently, a clause is more likely to be retained if it has facts in its bodies, so this clauses should have a higher priority than ordinary clauses. The following algorithm uses such heuristic order that
Chapter 4. Optimizations of the Incremental Algorithm and Partial Instantiation

attempts to reduce the times Subroutine Remove will be called.

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

1. Initialize $R$ to all the facts among $Cl_1, ..., Cl_n$, 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 10 Suppose the six clauses of $P$ and $P'$ in Examples 2 and 3 are to be inserted. Clause 6 is the first one considered. Since IncrOpt does not add Clause 6 to the DC-Graph, Clauses 1 and 7 are added to the set $S$ and inserted in the next iteration of IncrOptOrder. While Clause 1 is added to the DC-Graph, Clause 7 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 6 is inserted after Clause 1, then node $C$ created during the insertion of Clause 1 will need to be removed. IncrOptOrder relies on inserting facts first and on Step 2b to prevent all these happening.
4.2.4 Only Facts Ordered

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 an arbitrary order.

**Algorithm IncrOptFact** Let $Cl_1, ..., Cl_n$ 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.

Since facts are the basic sources that cause removals in a DC-Graph, IncrOptFact might considerably improve the performance by considering facts first. Furthermore, IncrOptFact does not have much overhead because it is very easy to distinguish facts from ordinary clauses.

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

4.3 Rules for Cutting Redundant Nodes

Thus far we have studied the optimization methods for Algorithm Incr. But recall that our final goal is to expand the partial instantiation trees efficiently for computing minimal and least models. In this section we will study how to optimize partial instantiation trees.

As described in Chapter 3.1.1, partial instantiation can be viewed as expanding and processing nodes of partial instantiation trees. For each conflict-set unifier $\theta$ of a node in a partial instantiation tree, there is a child node processing $P \cup P\theta$. If a newly generated node $N$ is identical to one of its ancestors, node $N$ is redundant and could be cut off.
Chapter 4. Optimizations of the Incremental Algorithm and Partial Instantiation

Obviously if there is a way to predict redundant nodes, we can save the computations on these nodes by not expanding them, and thus improving the efficiency of partial instantiation. By carefully examining the unifiers along expansion path in partial instantiation tree, we develop the following lemma which is very useful in avoiding redundant node expansion.

This lemma gives 3 sufficient conditions which are easy to implement. Without loss of generality, it assumes that substitutions in conflict-set unifiers are represented in solved form [34]. 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 $\theta$ respectively. We also use the notation $P \triangleright P_1$ to denote the fact that the node for program $P$ is the parent of the node for $P_1$, and $\theta$ is the conflict-set unifier, i.e. $P_1 = P \cup P\theta$.

**Lemma 7**

1. Given $P \triangleright P_1$ and $P_1 \triangleright P_2$, it is necessary that $P_2 = P_1$.

2. Given $P \triangleright P_1 \triangleright P_2$, and $P \triangleright P_3 \triangleright P_4$, $P_4 = P_2$ 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 $P \triangleright P_1 \triangleright P_2 \triangleright P_3$, $P_3 = P_2$ if $L(\theta_1) \cap R(\theta_2) = \emptyset$.

**Proof**

1. By definition, $P_2 = P_1 \cup P_1\theta$. Substituting $P_1 = P \cup P\theta$ into it, we get $P_2 = P \cup P\theta \cup P\theta$. Since $\theta$ is in resolved form, $\theta = \theta\theta$. Thus we have $P_2 = P \cup P\theta$, which is equal to $P_1$. 

2. By definition, we have $P_2 = P_1 \cup P_1 \theta_2$. Substituting $P_1 = P \cup P \theta_1$ into it, we get $P_2 = P \cup P \theta_1 \cup P \theta_2 \cup P \theta_1 \theta_2$. On the other hand, by definition, we have $P_4 = P_3 \cup P_3 \theta_1$. Substituting $P_3 = P \cup P \theta_2$ into it, we get $P_4 = P \cup P \theta_1 \cup P \theta_2 \cup \theta_2 \theta_1$. It is easy to verify that $\theta_1 \theta_2 = \theta_2 \theta_1$ when $L(\theta_1) \cap L(\theta_2) = \emptyset$, $L(\theta_1) \cap R(\theta_2) = \emptyset$, $R(\theta_1) \cap L(\theta_2) = \emptyset$. Therefore we have $P_4 = P_2$.

3. By definition, $P_3 = P_2 \cup P_2 \theta_1$. Substituting $P_2 = P_1 \cup P_1 \theta_2$, we get $P_3 = P_1 \cup P_1 \theta_2 \cup P_1 \theta_1 \cup P_1 \theta_2 \theta_1$. Since $L(\theta_1) \cap R(\theta_2) = \emptyset$, $P_1 \theta_2 \theta_1 = P_1 \theta_2$. So $P_3 = P_1 \cup P_1 \theta_2 \cup P_1 \theta_1 \cup P_1 \theta_2$. Then substituting $P_1 = P \cup P \theta_1$, we have $P_3 = P \cup P \theta_1 \cup P \theta_2 \cup P \theta_1 \theta_2$. On the other hand, by definition, $P_2 = P_1 \cup P_1 \theta_2 = (P \cup P \theta_1) \cup (P \cup P \theta_1) \theta_2 = P \cup P \theta_1 \cup P \theta_2 \cup P \theta_1 \theta_2 = P_3$. □

Applying the lemma above to tree expansion, some branches can be cut off, as shown in the figure below, when $\theta_1, \theta_2$ satisfy the corresponding conditions.

![Figure 4.1: Cutting Redundant Branches](image)

As an example, consider again the program $P$ discussed in Chapter 3.1.1. As shown in Figure 3.1, $P_1$, which is defined by $P_1 = P \cup P \theta_1$, has two child nodes – node 3 and
node 4, corresponding to the conflict-set unifiers $\theta_1$ and $\theta_2$. Then according to the first part of lemma 7, there is no need to expand the node $P_3 = P_2 \cup P_2 \theta_1$, because $P_3$ is identical to $P_1$. Look at node 7: since $\theta_7 = \theta_1$, according to the last part of Lemma 7, node 7 does not need to expand either. Applying Lemma 7 to other nodes, we can get the reduced partial instantiation tree as shown in Figure 4.2. As can be seen, 6 out of 11 nodes are cut off.

\[
\begin{align*}
\theta_1 &= \{X1 = a, Y1 = Y2\} \\
\theta_2 &= \{X1 = X2, Y1 = b\}
\end{align*}
\]

![Reduced Partial Instantiation Tree](image)

In this chapter, we have optimized Algorithm Incr by deleting clauses in self-sustaining cycles and ordering clauses to be inserted. We have also optimized partial instantiation by avoiding redundant nodes in partial instantiation trees. In next chapter, we will
present experimental results evaluating the effectiveness of these optimizations.
Chapter 5

Experimental Results

In the previous chapter, we have explored how Algorithm Incr and partial instantiation can be optimized. To evaluate the effectiveness of these optimization methods, we implemented the corresponding algorithms and then conducted several series of experiments, including the comparison of the three orders for newly inserted clauses, the comparison between Algorithm IncrOptFact and SizeOpt on both definite and disjunctive cases, and last but not least, the comparison of partial instantiation trees with and without cutting redundant nodes. The experiment results to be presented in this chapter will show that the optimization methods can bring about significant improvement in performance.

5.1 Experiment Overview of the Incremental Algorithms

For our experimentation, we implemented Algorithm IncrOpt (and thus trivially IncrOptArb), IncrOptOrder and IncrOptFact. We also implemented two versions of Algorithm SizeOpt. One is a straightforward encoding of the algorithm presented in Chapter 3.1.2. 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 and maintain 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 algorithm, 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 \( Cl \equiv A_1 \lor \ldots \lor A_m \leftarrow B_1 \land \ldots \land B_n \), 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 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-graph, 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 \( Q \).

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

5.2 IncrOptFact vs IncrOptOrder vs IncrOptArb

In this series of experiments, we compared the effectiveness of the heuristics described in Chapter 4.2. The following results are very representative of all the experiments we conducted. In this particular experiment, the input program \( P \) consists of 20 disjunctive clauses. All of them are of the following form:

\[ A_1 \lor \ldots \lor A_m \leftarrow B_1 \land \ldots \land B_n. \]
Here $m$ is less than 5, and $n$ less than 10, which means at most 5 atoms appear in the head of each clause, and at most 10 appears in the body. An atom is represented by an integer ranging from 1 to 15. The numbers of heads and bodies in each clause, as well as atoms in the clause, are randomly generated. The times below count the time taken for each algorithm to process $P$.

<table>
<thead>
<tr>
<th></th>
<th>IncrOptFact</th>
<th>IncrOptArb</th>
<th>IncrOptOrder</th>
</tr>
</thead>
<tbody>
<tr>
<td>time(ms)</td>
<td>3.5</td>
<td>3.6</td>
<td>150.6</td>
</tr>
</tbody>
</table>

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.

5.3 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 on different sets of clauses. Each clause set has 20 disjunctive clauses generated the same way as in section 5.2. Particularly clause set 1 is the same set used in section 5.2. For both algorithm IncrOptFact and SizeOpt, we report:

1) T1: the time taken to process the 20 clauses
2) N: the number of clauses deleted
3) T2: the time taken to compute the minimal models

4) T3: the total time, i.e., the sum of T1 and T2

<table>
<thead>
<tr>
<th></th>
<th>Clause Set 1</th>
<th>Clause Set 2</th>
<th>Clause Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IncrOptFact</td>
<td>SizeOpt</td>
<td>IncrOptFact</td>
</tr>
<tr>
<td>T1</td>
<td>3.54</td>
<td>0.33</td>
<td>3.65</td>
</tr>
<tr>
<td>N</td>
<td>19</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>T2</td>
<td>49.17</td>
<td>83.61</td>
<td>78.95</td>
</tr>
<tr>
<td>T3</td>
<td>52.71</td>
<td>83.94</td>
<td>82.6</td>
</tr>
</tbody>
</table>

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 in the table, the extra time is worth spending because IncrOptFact manages to delete more clauses than SizeOpt. This is all due to the fact that, as described in Chapter 4.1, IncrOptFact deletes all the clauses in self-sustaining cycles. SizeOpt deletes no clause here because for disjunctive programs, each clause has multiple heads, and thus making it hard to find an atom that doesn’t appear in all these heads. Consequently, the times taken for the two algorithms to find the (same collection of) minimal models differ by a wide margin. This 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% - 70% of the time taken by SizeOpt.

Due to lack of proper real-world deductive databases, the clause sets used in these series of experiments are randomly generated. The effect of Algorithm IncrOptFact on real-world deductive databases remains to be examined.
Chapter 5. Experimental Results

5.4 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.

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.

<table>
<thead>
<tr>
<th></th>
<th>IncrOptFact</th>
<th>SizeOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>processing time for 100 clauses (ms)</td>
<td>9.22</td>
<td>0.73</td>
</tr>
<tr>
<td>rules deleted</td>
<td>89</td>
<td>17</td>
</tr>
<tr>
<td>time to find least model (ms)</td>
<td>5.76</td>
<td>580.06</td>
</tr>
<tr>
<td>total time taken (ms)</td>
<td>14.98</td>
<td>580.79</td>
</tr>
</tbody>
</table>

5.5 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
Chapter 5. Experimental Results

instantiation trees efficiently. As described in Chapter 3.1.1, 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 3.2, the acid test of the effectiveness of our incremental algorithms is between the time taken for our incremental algorithms to process the clauses in $P\theta_j$. Given the results of the previous series of experiments, we expect IncrOptFact to outperform SizeOpt even more in the expansion of partial instantiation trees. As an example, we fully expand the instantiation tree of the program discussed in Chapter 3.1.2 using both algorithms. Note that this example was brought out first in other papers ([14],[15]) so it is rather representative.

By applying the heuristics of avoiding redundant node expansion discussed in Chapter 4.3, our algorithm only needs to process 5 nodes, as shown in Figure 4.2, when compared with 11 that would be needed otherwise. 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 - $P\theta_j$ for IncrOptFact (for all nodes), and ii) the time taken to find the least model.

<table>
<thead>
<tr>
<th></th>
<th>IncrOptFact</th>
<th>SizeOpt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node 1 (ms)</td>
<td>0.67/5.47</td>
<td>0.33/45.88</td>
</tr>
<tr>
<td>Node 2 (ms)</td>
<td>0.02/5.57</td>
<td>0.34/45.86</td>
</tr>
<tr>
<td>Node 3 (ms)</td>
<td>0.02/5.57</td>
<td>0.34/53.95</td>
</tr>
<tr>
<td>Node 4 (ms)</td>
<td>0.02/5.49</td>
<td>0.34/49.49</td>
</tr>
<tr>
<td>Node 5 (ms)</td>
<td>0.02/5.57</td>
<td>0.34/52.88</td>
</tr>
<tr>
<td>total (processing time/model solving time)</td>
<td>0.75/27.67</td>
<td>1.69/247.76</td>
</tr>
<tr>
<td>total (processing time + model solving time)</td>
<td>28.42</td>
<td>249.45</td>
</tr>
</tbody>
</table>

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. The total time taken to expand the 5 nodes by using IncrOptFact is merely only 10% of the time taken by using SizeOpt in this example.

5.6 Partial Instantiation Trees: With and Without Applying Lemma 7

As shown before, Lemma 7 provides three sufficient conditions to predict redundant nodes. This certainly saves us the processing time for expanding these redundant nodes in generating partial instantiation trees. However, applying Lemma 7 needs extra processing time to check at each node if any of the three conditions is satisfied. Taking both the gains and overhead into consideration, what will the overall performance be? To answer this question, again we use the example discussed in Chapter 3.1.1.

As we have already seen, by applying Lemma 7 to this example, the number of nodes in the partial instantiation tree can be reduced from 11 to 5. The following table shows the total processing time IncrOptFact takes to expand the 5 nodes using Lemma 7 vs the time it takes to expand the 11 nodes without using Lemma 7. Unlike the experiments we did above, the total processing time here includes the I/O time. i.e., reading in the logic program and print out the facts deduced.

<table>
<thead>
<tr>
<th>time(ms)</th>
<th>optimization</th>
<th>non-optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>108.53</td>
<td>130.67</td>
</tr>
</tbody>
</table>

As can be seen, regardless of the overhead needed, 25% of the total processing time can be saved by using Lemma 7 to cut down redundant nodes.
In this chapter, we can see that among the three orders for inserting new clauses, Algorithm IncrOptFact, i.e., processing the facts first, gives the best performance. When compared with the original algorithm SizeOpt, IncrOptFact works better in all the experiments we have conducted. Cutting the redundant nodes in partial instantiation trees can also bring about significant improvements. In the next chapter, we will report how we implement the whole framework of partial instantiation.
Chapter 6

Implementation Details of Partial Instantiation

We have implemented the framework of partial instantiation system and integrated it with the optimization methods developed in this thesis. The implementation is written in C language running under UNIX environment. It has roughly 3,000 lines of code. In this chapter, we will discuss the major modules and data structures in our implementation.

As introduced in Chapter 1, there is a unique least model for a definite database, and one or more minimal models for a disjunctive database. Due to the lack of a proper minimal model solver at this stage, our implementation only handles definite databases. But it is easy to extend the current framework to handle disjunctive databases when minimal model solver becomes available. All we need to do is to replace the least model solver with the minimal model solver. The interface remains the same.

6.1 General Picture

The system takes a logic program $P$ as input and returns the partial instantiation tree of $P$ along with all the derived facts. Depth-first strategy is chosen here to expand tree nodes for time and space considerations.

First let us look at what happens at each node. Suppose a node $N$ is expanded through unifier $\theta$, and the logic program associated with its parent is $P$.

1. Get new program $P_N = P \cup P\theta$
2. For new clauses in $P_N$, apply IncrOpt algorithm to it to get the reduced program $P'_N$.

3. Apply minimal model solver to $P'_N$, get T and F set.

4. From T and F, get the conflict unifier set.

5. Cut unifiers by Lemma 7 described in Chapter 4 to avoid redundant nodes.

6. Check if the current node is a leaf node. If yes, Stop. Otherwise for every $\theta$ in the conflict unifier set, a corresponding child node is created.

This expansion continues recursively until a node is a leaf node. A node is a leaf node when either of the following conditions is satisfied:

- no unifiers at this node.

- this node is a copy of one of its ancestors, i.e., the programs, T sets and F sets of the two nodes are identical.

Considering the major steps involved at each node, we can see that Step 2, i.e., applying IncrOpt algorithm, has been studied thoroughly in the previous chapters. The implementation of steps 1, 3, 4, 5 will be discussed step by step later in this chapter. Before that we need to first discuss the supporting data structures.

### 6.2 Data Structures

Following we will discuss the major data structures used in the system and the considerations behind them.
6.2.1 Term and Term Table

Term is the most basic element in a deductive database. The definition of term is recursive. A term can be a constant, a variable, or of the form $f(t_1, t_2, ..., t_n)$ where $f$ is a function name of arity $n$ and each $t_i$ is a term. Following are the fields in the structure that represents terms.

- **type**: an integer indicating the type of the term. 0: CONSTANT; 1: VARIABLE; 2: FUNCTION.
- **arity**: an integer.
- **name**: a string. We adopt the Prolog convention of denoting variables by strings of characters starting with an upper case letter and constant names by strings of characters starting with a lower case letter or integers.
- **subTerms**: pointer to the first parameter of a function term. This field is valid iff the term's type is FUNCTION and arity is not 0.
- **nextTerm**: pointer to the next term. It is used to link the rest of the parameters in a function term. Only when the term's type is FUNCTION and arity $\geq 2$ may this field be valid.
- **unifiedTerm**: an integer which is the index of some term that is derived by applying a unifier to this term. This field is used for replacement.

As an example, Figure 6.1 shows how the term $f(x, g(x, y))$ can be represented using the structure above.

All the terms are organized in a global table—term table. Actually each term is identified by its index number in term table. At the root node, term table contains all the terms in the original program. When a child node is created, new terms might be
Chapter 6. Implementation Details of Partial Instantiation

term: \( f(a, g(X, Y)) \)

![Diagram showing term structure](image)

Figure 6.1: The Structure of Term \( f(a, g(X, Y)) \)

generated via unification. The new terms are added to the end of the term table. After
the child node is done, these newly generated terms at this node can be thrown away
because they are not useful for other nodes.

As we can see, all the insertion and deletion operations of term table happen to the
end of the table. Thereafter we use stacks to represent it. A stack is an array of pointers
with a stack pointer points to the last currently active element. Any element behind
this pointer is a NULL pointer, and any element before this pointer is a pointer to the
corresponding structure (clause, atom or term). We choose this data structure because
of following reasons:

- Memory consideration: Compared to ordinary array, i.e., each element of the array
  is an actual structure, this data structure uses less memory. It allocates memory
  for an element only when the element becomes active. Otherwise, an element only
  takes four bytes for an pointer which is NULL. When an active element becomes
inactive, the memory allocated to it can be released.

- Efficiency consideration: Compared to data type link, this structure is more efficient as well as easily implemented. An element can be accessed directly by its index to the array. And when an element is inserted or deleted, what needs to be done is just to increase or decrease the stack pointer.

### 6.2.2 Atom and Atom table

An atom is of the form \( p(t_1, t_2, ..., t_n) \) where \( p \) is a predicate name of arity \( n \) and each \( t_i \) is a term. Following are the fields in the atom structure.

- name: a string of characters.
- arity: an integer;
- terms[MAXATOMARITY]: array of integers. The \( i \)th \((i < \text{arity})\) integer represents the index of the \( i \)th term in term table. In this way we only need to store one copy of any term.
- unifiedAtom: integer.

All the atoms are organized in a global atom table, which is very similar to term table. For the same considerations as term table, atom table is actually a stack. Each atom is identified by its index in the atom table.

### 6.2.3 Clause and Clause Table

A clause is a statement of the form

\[
A_1 \lor A_2 \lor ... \lor A_n \leftarrow B_1 \land B_2 \land ... \land B_n.
\]

where \( A_i \)'s and \( B_i \)'s are atoms. \( A_i \)'s are called the head of the clause, and \( B_i \)'s are called the body of the rules.
Chapter 6. Implementation Details of Partial Instantiation

Each atom in a clause is actually an integer corresponding to the atom's index in atom table. Considering when an old program P is expanded to a new program $P \cup P\theta$, we need to compare each clause in $P\theta$ to all the existing clauses in P to get the new clauses. To make this comparison faster, the atom indices in a clause head and in a clause body are kept by an increasing order.

- head_number: integer.
- heads[MAXHEAD]: indices to atom table.
- body_number: integer.
- bodies[MAXBODY]: indices to atom table.

Again all the clauses are organized in a global clause table. Similar to term table and atom term, clause table is a stack. Each clause is identified by its index in clause table.

6.2.4 Tree

A tree node contains the following fields:

- parent: pointer to its parent node.
- children: array of pointers to its children.
- childNo: index to the children array of its parent.
- uSet: pointer to the set of unifiers of this node.
- truthSet: pointer to the T set.
- falseSet: pointer the F set.
- clauseNum: index of the last active clause in program table.
- atomNum: index of the last active atom in atom table.
Chapter 6. Implementation Details of Partial Instantiation

- termNum: index of the last active term in term table.
- rules: the rule part of the DC-Graph in incremental program.
- atoms: the atom part of the DC-Graph in incremental program.

Looking into a tree node, we found that a lot of information in a tree node, such as T set, F set, and DC-Graph, are intermediate results during the processing of the node. Once the node is fully expanded, these pieces of information are not useful anymore. Considering this fact, we define most of the fields of a node as pointers so that the memory allocated to them can be released after the node has been done. This can save us a lot of memory.

So far, we have introduced the global data structures in our implementation. In the rest of this chapter, we will talk about the major steps described in section 6.1 on a step-by-step basis.

6.3 Generating New Clauses

One important step while expanding a node is: given a program $P$ and a unifier $\theta$, how to get the new program $P \cup P\theta$? A transformation of this problem is: given $P$ and $\theta$, how to get the new clauses by applying $\theta$ to $P$. Once we have the new clauses, we can get the new program by simply adding new clauses to the end of the original program $P$.

The following three steps are needed to get the new clauses generated by $P\theta$.

- For every term $t$ in term table, get its unified term $t\theta$. Three possible situations might happen: $t\theta$ is $t$ itself, or it is some other existing term in the term table, or it is a totally new term. In the third case, the new term is added to the end of term table. Recall that for any term $t$, there is a field called "unifiedTerm". This field
is used to store the index of term $t\theta$.

- For every atom $a$ in atom table, get its unified atom $a\theta$. $a\theta$ can be simply achieved by replacing its terms with the corresponding unified terms. Similarly, $a\theta$ could be itself, or some other existing atom in atom table, or a totally new atom. The "unifiedAtom" field in the atom structure stores the index of atom $a\theta$. In the third case, the new atom is added to the end of the table.

- For every clause $Cl$ in $P$, get its unified clause $Cl\theta$. $Cl\theta$ can be simply achieved by rewriting each atom in $Cl$ with its unified atom. $Cl\theta$ is compared with every clause in clause table. If it is different with all of the clauses, it is a new clause and should be added to the end of clause table. Otherwise it is ignored.

It can be seen that a lot of comparisons are required here. In order to check if a term/atom/clause is new or not, we need to compare it with each and every term/atom/clause in the term/atom/clause table. As we discussed before, the term/atom/clause table throw away all the obsolete elements when a node is done. In this way we minimize the size of these tables, thus making the comparison more efficient. Furthermore, we keep the atoms in a clause by an increasing order. When we compare two clauses $Cl_1$ and $Cl_2$, we only need to compare the $n$th ($1 \leq n \leq N$, $N$: the total number of atoms in $Cl_1$) atom in $Cl_1$ with the $n$th atom in $Cl_2$, saving the trouble of comparing the $n$th atom in $Cl_1$ with each and every atom in $Cl_2$.

### 6.4 Least/Minimal Model Solver

Least/Minimal model solver takes a definite/disjunctive program $P$ as input and returns the $T$ set and $F$ set of $P$. As discussed before, by applying incremental algorithm IncrOpt, the least model of a definite program can be easily acquired in the following way: for any
atom, it is in T set if it is the head of some retained clause; otherwise it is in F set. So the least model solver module is rather simple to implement. All it does is to collect all the heads of retained clauses and put these atoms in T set. The rest of the atoms are put in F set.

Due to the lack of a proper minimal model solver at this stage, our implementation cannot handle disjunctive programs thus far. However, once minimal model solver becomes available, our system can be very easily extended to handle disjunctive cases. This is because we took disjunctive databases into our consideration when we designed the data structures and implemented other parts of the system. For example, the structure of a clause includes an integer field which indicates the number of atoms in the head of the clause. In definite cases, this number remains 1, which means that only one head in every clause. For a disjunctive clause, the number may be equal or greater than 1. There is no need to modify the clause structure itself. So the extension from definite cases to disjunctive cases can be simply achieved by replacing the least model solver with any minimal model solver. All the other parts remain unchanged.

6.5 Unification

**Definition** [34] (*Unifier*). Two atoms (or expressions) A and B are unifiable if there is a substitution \( \theta \) such that \( A\theta = B\theta \). The substitution \( \theta \) is called a unifier for A and B. It is called a most general unifier if for each unifier \( \eta \) for A and B there exists a substitution \( \gamma \) such that \( \eta = \theta \gamma \).

More straightforwardly, the unification problem can be represented as the solution of a system of equations. That is, for a set of equations

\[ t_j' = t_j'', \quad j=1, \ldots, k \]
a unifier $\theta$ is any solution which makes all pairs of terms $t_i', t_j^*$ identical simultaneously.

### 6.5.1 Algorithm UNIFY: an Efficient Unification Algorithm

In partial instantiation, getting the unifiers for the T set and F set is the key step involved at each tree node, and thus the performance of unification algorithm affects in a crucial way the global efficiency. Many studies have been made to find efficient unification algorithms [35] [36] [37] [34] in literature. We found that the algorithm developed by Alberto Martelli and his colleagues in [34] was proved to have a better performance in all cases as compared with other well known algorithms. This is the algorithm we choose and implement. In the following we call it Algorithm UNIFY.

To understand the algorithm, we should first understand the following concepts.

- **Multiequation.** A multiequation can be seen as a way of grouping many equations together. To represent multiequations we use the notation $S = M$ where the left-hand side $S$ is a nonempty set of variables and the right-hand side $M$ is a multiset of nonvariable terms.

- **Common part.** The common part of a multiset of terms $M$ is a term which, intuitively, is obtained by superimposing all terms of $M$ and by taking the part which is common to all of them starting from the root.

- **Frontier.** The frontier of a multiset of terms is a set of multiequations, where every multiequation is associated with a leaf of the common part and consists of all subterms (one for each term of $M$) corresponding to that leaf.

- **Compactification.** All multiequations belonging to the same equivalence class should be transformed into single multiequations by taking the union of their left- and right-hand sides. This merging process is called compactification.
For instance, given the multiset of terms

\[ f(x_1, g(a, a, f(x_5, b))), \]
\[ f(h(c), g(x_2, f(b, x_5))), \]
\[ f(h(x_4), g(x_6, x_3)). \]

the common part of these three terms is

\[ f(x_1, g(x_2, x_3)). \]

The frontier is

\[ \{x_1\} = (h(c), h(x_4)), \]
\[ \{x_2, x_3\} = (a), \]
\[ \{x_3\} = (f(x_3, b), f(b, x_3)). \]

**Algorithm UNIFY [34]**

**input** U – the multiequations that needs to be unified

**output** T – the most general unifier of U. Initialized as empty.

1. Select a multiequation \( S = M \) of U such that the variables in \( S \) do not occur elsewhere in U. If no such multiequation STOP with failure.

2. if \( M \) is empty

    then transfer this multiequation from U to the end of T.

else

    i. compute the common part \( C \) and the frontier part \( F \) of \( M \). If \( M \) has no common part STOP with failure.

    ii. transform U using multiequation reduction on the selected multiequation and compactification.
iii. transfer the multiequation $S = (C)$ from $U$ to the end of $T$.

3. if $U$ is empty, stop with success, else go back to step 1.

$$\square$$

6.5.2 Implementation of Algorithm UNIFY

In our implementation, we represent a unifier as a set of special multiequations. For each multiequation, its left side is a set of variables, and its right side is a term. We keep the variables on the left hand side of any multiequation by their alphabet order for efficiency reasons (to be discussed in next section).

Given a multiterm $M$, the computation of its common part $C(M)$ and frontier part $F(M)$ is a recursive.

1. If all the terms in $M$ are the same constants $c$, then $F(M) = c, C(M) = \emptyset$.

2. else If $\exists t \in M$, $t$ is a variable then

   (i) $C(M) = t$

   (ii) the left-hand side of $F(M)$ is the set of all variables in $M$ and the right-hand side of $F(M)$ is the multiset of all terms in $M$ which are not variables.

3. else if all the terms in $M$ has same function symbol $f$ and same arity $n$ then

   (i) let $M_i = \{ t | t \text{ is the } i \text{.th term in } T \text{ where } T \in M \}, i = 1, \ldots, n$

   (ii) $F(M) = F(C(M_1), \ldots, C(M_n))$

   (iii) $C(M) = \cup_{i=1}^n F(M_i)$

4. else failure.

Compaction involves a lot of term comparisons. Term comparison again is a recursive processing. To compact two multiequations $S_1 = M_1$ and $S_2 = M_2$, we first:
compare if there exists any variable \( v \) which appears in both \( S_1 \) and \( S_2 \). If yes, the merged multiequation is \( S = M \) where \( S = S_1 \cup S_2 \), \( M = M_1 \cup M_2 \). In order to make comparisons faster, we order the variables in \( S_1 \) and \( S_2 \) by their alphabet order before we start compactification.

6.6 Cutting Unifiers

Recall that in Chapter 4.3, Figure 4.1 shows three situations when a unifier leads to redundant nodes. These unifiers should be removed from the unifier set of a node in order to avoid redundant node expansion.

This step is rather straightforward, involving a lot of comparisons among unifiers. As discussed before, in every multiequation of a unifier, all the variables on the left hand side are kept by their alphabet order. This makes the comparison of unifiers easier and more efficient.

In this chapter, we discussed the implementation of the framework of partial instantiation. We first introduced the global data structures and the considerations behind them. Then we discussed, on a step-by-step basis, all the major steps required at expanding a tree node. In the next chapter, we will present the conclusion of thesis, including thesis summary and future works.
Chapter 7

Conclusions

7.1 Thesis Summary

Methods to compute minimal models of disjunctive programs are becoming increasingly important as an intermediate step in query evaluations. Partial instantiation is a newly developed method to compute minimal models on an “instantiation-by-need” basis. The objective of this thesis 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 to reduce the size of deductive databases. Incr aims to be incremental in the sense that when new clauses are added to the database, only these new clauses need to be handled for getting the new set of reduced clauses. A DC-Graph that organizes the deleted clauses is used to achieve this incrementality. Incr is formally proved to be incremental.

We have also optimized Incr to IncrOpt which deletes clauses in self-sustaining cycles. Compared to Incr, IncrOpt not only deletes more clauses, but also solves the program directly when applied to definite programs. For further optimization, we examined different orders for the clauses to be inserted, which leads to several algorithms. Experimental results indicate that IncrOptFact, which handles the facts ahead of ordinary clauses, gives the best performance. Most importantly, when compared with the original algorithm SizeOpt, IncrOptFact can give very significant improvements in run-time efficiency.
Besides the development of Incr, we have also optimized the expansion of partial instantiation trees. We have designed three rules which cuts down redundant node expansions. Experimental result shows that this optimization further improves the efficiency of partial instantiation.

Finally, we have implemented the whole framework of partial instantiation, integrated with the optimization methods developed in this thesis. The implementation takes a deductive database as input and returns its complete partial instantiation tree.

7.2 Future Work

Techniques to further optimize the run-time performance of partial instantiation are to be explored. Following are some open problems in this direction.

7.2.1 Cutting Redundant Nodes

In this thesis, we have developed three rules to cut down redundant nodes. These rules only cuts down some of the redundant nodes, not all of them. In extreme cases, redundant nodes could make the process of expansion very inefficient. Thus a deeper study on redundant nodes – what causes them and how to prevent them from being generated—will greatly improve the efficiency of partial instantiation.

7.2.2 Order for Node Expansion

A partial instantiation tree can be generated in various orders which may perform differently. Two general strategies used in expanding a tree are breath-first and depth-first. Which method is more preferable in generating partial instantiation trees? Furthermore, at a given node with more than one children, the different orders of expanding these
children may also have an effect on performance. Which child should be chosen to expand first? Does there exist a best order? If yes, how to find it? These are interesting questions yet to be answered.

7.2.3 Tree Maintenance

Every deductive database is associated with a partial instantiation tree. The costs of generating these trees are usually expensive. This leads us to the idea of dynamically maintaining a tree when the original database undergoes changes. In other words, instead of generating a new tree each time when a database is updated, we want to modify the original tree and make it adapt to the updated database. This problem can be split into four smaller problems: a) adding a new fact; b) adding a new clause; c) removing an existing fact; and d) removing an existing clause. More research can be done to find out what kind of modifications should be made to the original tree when one or more of the above situations occur.

7.2.4 Partial Tree

In situations where it is not desirable or too costly to generate an entire partial instantiation tree, we need to generate part of the tree selectively. There are different criteria for selection. For example, generating as many nodes as possible when given a fixed amount of time and memory, or generating as many facts as possible when given fixed amount of nodes, etc. Which criteria is more reasonable? How to satisfy a certain criteria? These questions are open for further research.
Bibliography


