Optimizing the Computation of Stable Models using Merged Rules

Thesis Report

by

Veena S. Mellarkod

May 2002

ACKNOWLEDGEMENTS

I have been fortunate enough to have all the support I needed to finish my thesis work. First and foremost, I would acknowledge the real debt of gratitude that I owe to my parents and sister for always being there for me. Thanks Mom and Dad! Thank you Vidhya for having confidence in me, when I myself did not.

My special thanks go to my professor and guru, Dr. Michael Gelfond, for showing utmost patience in helping me. It was really fun working with you. Thank you Dr. Gelfond for teaching me, encouraging me, and making me what I am today. I express my special thanks to Dr. Richard Watson and Dr. Larry Pyeatt for serving on my committee and for their detailed reviews and helpful suggestions.

I should thank all my lab mates for providing a wonderful working environment in the lab. Thank you Monica for working with me and helping me in all phases of the work. It would not have been possible without your help and encouragement. You believed more than me that it would work. Thanks for always being there for me and listening to all the craziest ideas I get. This thesis work is one such crazy idea that you listened to on a Sunday morning, and if not for you, it would have remained so. Thank you Marcy for all the help you have done for me. Thanks for taking time from your work to teach and explain me the loopholes and dead ends in my ideas. If not for you and Monica this would not have come true. Thank you Greg for making me laugh; you always have helped me forget my tensions and concentrate on my work more.

Last but not the least, I would like to thank all my friends for giving me support and encouragement, for waiting late in the night while I was working to drive me safely back home, for helping me organize my defense and for everything else.

TABLE OF CONTENTS

ABSTRACT

Recently, logic programs under the stable model semantics, have emerged as a new paradigm for declarative programming. In this new approach, a logic program is used to represent the knowledge of the domain, and various tasks are reduced to computing the stable models of this program. This paradigm has been successfully used in a wide range of applications including planning, diagnostics, graph problems, etc. The basic algorithm for computing stable models is implemented by several efficient systems. The most efficient implementation to date is called *Smodels*. Even though *Smodels* was demonstrated to be capable of solving several large industrial problems, there are some simple logic programs for which *Smodels*′ performance is unexpectedly slow. This problem is not related to the implementation. Rather, it is the result of the *one rule at a time* inference used by the basic algorithm.

The goal of this work is to improve the efficiency of the basic algorithm extending the set of inference rules with a new rule called the *Extended Evaluation Rule* (*EER*). *EER* efficiently retrieves information spread across several rules of a program. An algorithm, *new_smodels*, was developed incorporating the *EER*. A system *Surya*, based on the *new_smodels* algorithm was implemented. It was found that the *EER* considerably improves the efficiency of the system.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The main goal of this chapter is to describe the problem of optimizing the computation of stable models, and to propose a solution which uses *merged* rules. The chapter is structured as follows : First, background information necessary to understand the subject is presented, then the problem we intend to solve is described, and finally a sketch of the solution proposed in this thesis is discussed.

1.1 Background

Programming languages can be divided into two main categories, *algorithmic* and *declarative*. Programs in algorithmic languages describe sequences of actions for a computer to perform, while declarative programs can be viewed as collections of statements describing the objects of a domain and their properties. This set of statements is often called a *knowledge base*. The semantics of a declarative program Π is normally given by defining its models, i.e., possible states of the world compatible with Π. The work of computing these models, or consequences, is often done by an underlying inference engine. For example, Prolog is a logic programming language that has such an inference engine built into it. The programmer does not have to specify the steps of the computation and can therefore concentrate on the specifications of the problem. It is this separation of logic from control that characterizes declarative programming [7, 10, 9].

Declarative programs need to meet certain requirements. Some of these requirements are[7]:

- The syntax should be simple and there should be a clear definition of the meaning of the program.

- Knowledge bases constructed in this language should be elaboration tolerant. This means that a small change in our knowledge of a domain should result in a small change to our formal knowlegde base.[11]

- Inference engines associated with declarative languages should be sufficiently general and efficient. It is often necessary to find a balance between the expressiveness of the language and the desired efficiency.

One such declarative language is $A-Prolog$ [7], a logic programming language under the answer set semantics [8]. The syntax of $A-Prolog$ is similar to Prolog. The following example of a program in $A-Prolog$ will be used through the introduction. Precise definition will be given in section 2.1.

Example 1.1 *Consider the program* Π *below:*

$$q(a).$$

$$q(b).$$

$$p(X) :- q(X).$$

Π *consists of three rules defining properties* $\{p,\ q\}$ *of objects* $\{\ a,\ b\ \}$. X *is a variable which is substituted or replaced by the objects in the program during its evaluation.*

The $A\!-\!Prolog$ language has the ability to represent a wide variety of problems such as reasoning with incomplete knowlegde and the causal effects of actions [3]. There are currently several inference engines for computing the answer sets of A-Prolog programs. Some of them are *Smodels* [14], *DLV* [1], *Romeo* [18], etc. The efficiency of these engines has led to some important applications including the use of *Smodels*, in the development of a decision support system for the space shuttle[2]. Other important applications are wire routing and satisfiability planning [6], encoding planning problems [4] and applications in product configuration [15], etc.

The *smodels* algorithm is a standard algorithm used for the computation of answer sets or stable models of a program. The *Smodels* system is one of the state-of-the-art implementations of the *smodels* algorithm. The System has a two level architecture. The frontend called *lparse* [16], takes a program with variables and returns a ground program by replacing all variables by constants in the program.

Example 1.2 *The grounding of program* Π *shown in example 1.1 would result in the* *program:*

$$q(a).$$

$$q(b).$$

$$p(a) :- q(a).$$

$$p(b) :- q(b).$$

In reality, *lparse* does more than just replacing variables by constants, but for simplicity sake we do not discuss its other functionalities here. The second part of the *Smodels* system is the inference engine *smodels*. It takes the ground program output by *lparse* and computes the stable models of the program.

The system *Smodels* has extended the language of A-Prolog with choice rules, cardinality rules and weight rules [12]. These rules increase the expressive power of the language in the sense that programs can be written more concisely using these extended rules. Here is an example of a cardinality rule in smodels language.

Example 1.3 *Consider a rule r,*

$$h :- 2\{\ a, b, c\ \}.$$

h is called the head of r and $2\{\ a, b, c\ \}$ *is called the body of r. The literals in the body of r are* $\{a, b, c\}$*. The rule is read as follows : "h is true if at least 2 literals from the body of r are true." Here 2 is called the lower bound of the rule.*

The *smodels* algorithm uses inference rules to compute the stable models or answer sets of a program. These inference rules play an important role in the efficiency of the algorithm. There are four important inference rules in the smodels algorithm. Given a set S of ground literals and a program Π,

1. If the body of a rule r, in Π, is satisfied by S then add the head of r to S.

2. If an atom a is not in the head of any rule in Π, then *not* a can be added to S.

3. If r is the only rule of Π with h in the head and $h \in S$ then the literals in the body of r can be added to S.

4. If h is in the head of rule r, in Π, *not* $h \in S$, and all literals in the body of r except l_i belong to S, then add *not* l_i to S.

Consider the following example which demonstrates the use of inference rule #1.

Example 1.4 *Let $S = \{a, b\}$ be a set of literals and rule r be of the form,*

$$h :- \ 2\{\ a, b, c\ \}.$$

Since literals a and b are true in S, at least two of the literals in the body of r are true in S; therefore, the body of rule r is satisfied by S. By the inference rule #1 we can conclude h and add it to S.

1.2 Problem Description

Before describing the problem, to facilitate its understanding, it would be helpful to explain how we became aware of it. At the beginning of year 2001, Vladimir

Lifschitz posed a *New Year's party* problem for the members of the Texas Action Group(TAG) [19], to solve. The *New Year's party* problem [22] consisted of finding a suitable seating arrangement for guests in a New Year's party, where the following two conditions needed to be satisfied. Guests who liked each other must be seated at the same table and guests who disliked each other must be seated at different tables. The number of tables, number of chairs around each table, and number of guests invited should be given as inputs to the program.

There were 23 solutions developed by TAG members. Most of the solutions used the *Smodels* language and the *Smodels* system to compute the stable models or answer sets of the program. Once the solutions were posted, the TAG group became involved in comparing these programs with respect to programming methodology and efficiency. Two programs (let us call them Π_1 and Π_2) caught the interest of the members of TAG in Austin[20] and TTU[21]. Both Π_1 and Π_2 were very similar. The only difference was that Π_1 had an extra rule which was not present in Π_2. The extra rule, r, was redundant to the program as the information given by r was already present among other rules of Π_1. This was the reason r was not present in Π_2. We found that Π_1 was far more efficient than Π_2 because of this rule. Though r was redundant, it helped in reducing the search space for computing the stable model(s) of Π_1. Computing models for Π_2 was slower because the information stated explictly by r, in Π_1, was *distributed* among several rules of Π_2.

The fact that a single rule, though a redundant one, could make such a difference in efficiency was important and pointed to serious difficulties for the programming methodology to be applied. It implies that programmers need to think and include all rules which gives information already present among other rules. It also means that the programmer needs to know which rules would make the program more efficient, and involves a deep understanding of the underlying smodels algorithm and implementation. Besides adding extra burden to programmers, this leads to less declarative programs, in the sense that a programmer's work is not just specifying the problem, but finding ways to improve efficiency based on the implementation of the inference engine. It would be preferable that the inference engine would automatically infer such information from the program, rather than requiring the programmer to write redundant rules.

The least efficient behaviour of Π_2 led us to the hypothesis that most of the inference rules of the smodels algorithm involved single $A-Prolog$ rules. This seemed the most straightforward explanation for the slower computation of the models of Π_2 by the system. If the inference rules could take into account information distributed among different rules in the program, then the number of inferences would be substantially higher. The following example illustrates this idea.

Example 1.5 *Consider rules r_1 and r_2 of a program Π,*

$$r_1 \quad h :- 2\{\ a, b, c\ \}.$$

$$r_2 \quad h :- 2\{\ not\ a, not\ b, not\ c\ \}.$$

It is easy to show that any answer set X of Π contains h. Consider two cases : (a) At least two of the atoms $\{a, b, c\}$ belong to X. In this case, $h \in X$ by r_1. (b) There are two atoms, say a, b, not belonging to X, then $not\ a, not\ b \in X$ and $h \in X$ by r_2. Notice that this reasoning, though simple, requires a simultaneous arguement about TWO rules of the program. The current version of smodels can not do such a reasoning. As a result h will be added to X after multiple tries and errors which substantially slows the performance.

It becomes clear that efficient implementation depends directly on obtaining as much information as early as possible and to do that, we need to expand the collection of inference rules of *smodels*. This is the problem we are interested in addressing in this thesis.

1.3 Proposed Solution

The work presented in this thesis consists of adding a new inference rule called the *Extended Evaluation Rule (EER)* to the *smodels* algorithm, in order to *merge* the information distributed among different rules of the program. This inference rule is applied to programs containing choice or cardinality rules. Given such a program Π, the EER inference rule consists of two steps:

1. Expand program Π, by adding a new rule $merge(R)$, where R is a set of cardinality rules of Π with same head. The construction guarantees that, *"A set of literals S satisfies R iff S satisfies $R \cup merge(R)$"*.

2. Check if the body of $merge(R)$ is satisfied by all stable models of Π containing S. If so then expand S by h.

Consider again example 1.5. Rules r_1 and r_2 of Π can be evaluated or merged into a new rule r_3 of the form : $h :- 3\{\ a, b, c, not\ a, not\ b, not\ c\ \}$. It is easy to see that body of r_3 is satisfied in all models of the program containing S and hence h can be derived.

The efficiency of EER depends on the efficient implementation of *merge* and the efficient checking of condition of clause (2) above. The construction of merge will be discussed in Chapter III. We will also show that, to efficiently perform the second step of EER, it is sufficient to do the following:

a. Compute the number of *complementary pairs* [1], *cp*, in the body of $merge(R)$;

b. If the lower bound of $merge(R)$ is less than or equal to *cp* then the body of the rule is satisfied by any stable model containing S.

[1]We call the pair $\{a,\ not\ a\}$, a complementary pair.

As we can see, this rule allow us to easily make this conclusion for literal h from r_3.

The contribution of this thesis is the development of a new algorithm called *new_smodels* algorithm incorporating the Extended Evaluation Rule, and implementation of a system based on the *new_smodels* algorithm. This thesis work includes:

1. Implementation of the system *Surya*, for computing stable models of a program based on *new_smodels* algorithm.

2. Experimental investigation of the efficiency of the EER inference rule.

3. Proof that the new inference rule maintains the *sound* and *complete* nature of the inference rules.

4. Proof of correctness of the *new_smodels* algorithm.

This thesis is organized in the following manner. Chapter II presents the syntax and semantics of the language $\mathcal{SL}$, a subset of the input language of *lparse*. Chapter III presents both the *smodels* and *new_smodels* algorithms. It discusses their similarities, and differences, and introduces the new inference rule, EER, added to *new_smodels* algorithm. Chapter IV presents the experimental results of the *Surya* system compared to the *Surya*$^-$ system, where *Surya*$^-$ is obtained from dropping the implementation of EER from *Surya*. Chapter V gives the conclusions and future work.

CHAPTER II

SYNTAX AND SEMANTICS

SMODELS is a system for answer set programming developed by Ilkka Niemela et al. [14]. It consists of **smodels**, an efficient implementation of the stable model semantics for normal logic programs, and **lparse**, a front-end that transforms user programs with variables into a form that smodels understands. The input language of lparse, extends A-Prolog with new rules, such as cardinality rules and choice rules [13]. Let us now define the syntax of $\mathcal{SL}$, a subset of the input language of $lparse$.

2.1 Syntax

The syntax of $\mathcal{SL}$ is determined by a signature $\Sigma = < C, V, P, F >$ where C, V, P and F are collections of object constants, variables, predicate symbols and function symbols respectively. A term of Σ is either a variable, a constant, or an expression $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms, and f is a function symbol of arity n. An *atom* is of the form $p(t_1, \ldots, t_n)$ where p is an n-ary predicate symbol, and $t_1, \ldots, t_n$ are terms. A *simple literal* is either an atom a, or its negation *not a*. Simple literals of the form *not a*, are called *not-atoms*, or *negative literals*. The symbol *not*, denotes a logical connective known as *default negation*, or *negation as failure*. The expression *not a* is read as *"there is no reason to believe in a." Ground* expressions are expressions which do not contain variables. Ground terms are represented by lower case letters, and variables by upper case letters.

A *simple rule* of $\mathcal{SL}$, is a statement of the form:

$$h :- \; l_1, \ldots, l_n. \tag{2.1}$$

where h is an atom, and $l_1, \ldots, l_n$ are simple literals. Atom h is called the head of the rule, and $l_1, \ldots, l_n$ constitute the body of the rule. Both the head and the body of the rule may be empty. If the body is empty, i.e., $n = 0$, the rule is called a $fact$, and we write it as h.

Simple rules of $\mathcal{SL}$ correspond to the rules of A-Prolog. To introduce the extended rules of $\mathcal{SL}$, we need the following definitions:

A *conditional literal* of $\mathcal{SL}$ is an expression of the form:

$$l_0 : l_1 : \ldots : l_n \tag{2.2}$$

where l_0 is a simple literal called the *literal part* of (2.2) and $l_1, \ldots, l_n$ are atoms called the *conditional part* of (2.2). There are some restrictions on the use of conditional literals. They will be dealt with at the end of this section.

A *constraint literal* of $\mathcal{SL}$ is an expression of the form:

$$lower\{l_1, \ldots, l_n\}upper \tag{2.3}$$

where lower and upper are arithmetic expressions called the *lower bound* and *upper bound* of the literal, and $l_1, \ldots, l_n$ are either simple or conditional literals. The lower and upper bounds need not be specified (if the lower bound is omitted, it is understood as zero and if the upper bound is omitted it is understood as infinity). If C is a constraint literal of the form (2.3), by lit(C) we mean $\{l_1, \ldots, l_n\}$.

A *cardinality constraint rule* of $\mathcal{SL}$, is a statement of the form:

$$l_0 :- \ l_1, \ldots, l_n. \tag{2.4}$$

where l_0 can either be empty, an atom, or a constraint literal, such that $lit(l_0)$ contains no negated literals. The literals in the body of (2.4), may be simple, conditional, or constraint literals. Obviously, simple rules of $\mathcal{SL}$ are just a special case of cardinality constraint rules.

A *logic program* is a pair $\{\Sigma, \Pi\}$, where Σ is a signature and Π is a collection of cardinality constraint rules over Σ.

The standard implementation of *smodels*, places some restrictions on logic programs it can reason with. To describe these restrictions we need the following definitions.

The collection of rules of a logic program Π whose heads are formed by a predicate p, is called the *definition* of p in Π. A predicate p is called a *domain predicate* w.r.t. a logic program Π, if:

a. the definition of p in Π has no negative recursion and

b. there exists no rule $r \in \Pi$, such that the head of r is a constraint literal C, and a simple literal l formed by p either:

   (i) belongs to $lit(C)$, or

   (ii) is the literal part of a conditional literal in $lit(C)$.

When using conditional literals, we need to distinguish between the *local*, and *global* variables in a rule. Given a rule r, which contains conditional literals, a variable is local to the occurence of a conditional literal in r, if the variable does not appear in any simple literal in r. All other variables are global.

Example 2.1 *Given a rule r,*

$$a :- \ 1\{p(X, Y) : q(Y)\}, r(X).$$

variable X is global and variable Y is local. Similarly, for

$$2\{p(X, Y) : q(X) : r(Y)\} :- \ s(Y).$$

Y is global and X is local.

The following are the restrictions in the implementation of smodels :

1. *The conditional part of any conditional literal must consist of only atoms formed from domain predicates.*

2. *Every local variable in the literal part of a conditional literal must appear at least once in its conditional part.*

3. *The programs are domain restricted in the sense that every variable in a rule r, must appear in an atom formed by a domain predicate in the body of r.*

From now on, by a program we mean a program of $\mathcal{SL}$ satisfying the above conditions.

Having defined the syntax of $\mathcal{SL}$, we are ready to define its semantics.

2.2 Semantics

This definition is done in two steps. First, we introduce a series of operations to transform an arbitrary program Π into a ground program $ground(\Pi)$. Second, we define the semantics of ground(Π). The semantics of the ground(Π), will be viewed as the semantics of program Π.

Let Π be an arbitrary program over a signature Σ. Let Π_g be the result of replacing all of the global variables of Π by ground terms of Σ. If Π_g is simple, i.e. a program consisting of simple literals, then $ground(\Pi) = \Pi_g$.

Example 2.2 *Consider* Π_0 *consisting of the following simple rules :*

$$d(a).$$

$$d(b).$$

$$p(X) :- q(X).$$

$$r(X) :- p(X).$$

the program $ground(\Pi_0)$ *is given below:*

$$d(a).$$

$$d(b).$$

$$p(a) :- q(a).$$

$$p(b) :- q(b).$$

$$r(a) :- p(a).$$

$$r(b) :- p(b).$$

To give the semantics for a simple program, we need to introduce some terminology. If S is a set of atoms, we say that S *satisfies* an atom a, $S \models a$, if $a \in S$ and S *satisfies not* a, $S \models not\ a$, if $a \notin S$.

A set of atoms, S, satisfies a simple rule r, if the head of r belongs to S whenever the body of r is satisfied by S. If the head of a rule is empty, then S satisfies the rule when at least one of the literals in the body is not satisfied by S. S satisfies a simple program Π, if it satisfies all of the rules of $ground(\Pi)$.

Given a set of atoms S and a simple rule r of the form :

$$h :- a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m. \tag{2.5}$$

we define the reduct of r with respect to S, r^S, as follows:

$$r^S = \begin{cases} \emptyset & \text{if } b_1, \ldots, b_m \cap S \neq \emptyset, \\ h :- a_1, \ldots, a_n. & \text{otherwise.} \end{cases}$$

we define the reduct of a program Π with respect to a set of atoms S, Π^S, as:

$$\Pi^S = \{r^S \mid r \in \Pi\}. \tag{2.6}$$

Definition 2.1 *The deductive closure of* Π *is the minimal set of atoms that satisfies* Π*.*

Definition 2.2 *A set of ground atoms* S *is a stable model of a program* Π *iff* S *satisfies all of the rules of* Π *and* S *is the deductive closure of* Π^S*.*

Example 2.3 *The program* Π_0 *be:*

$$p(a).$$

$$q(X) :- p(X).$$

has a unique stable model $S = \{p(a),\ q(a)\}$*.*

The program Π_1*,*

$$p(a) :- p(a).$$

has one stable model $S = \{\ \}$*.*

It can be shown that a program without default negation has exactly one stable model. The following are some examples of programs with default negation.

Example 2.4 *The program* Π_2*,*

$$p :- \ not\ q.$$

has one stable model $S = \{p\}$ *since*

$$\Pi^S = \{p :- \ \}$$

and S *is the deductive closure of* Π^S*.*

The program Π_3*,*

$$p :- \ not\ q.$$

$$q :- \ not\ p.$$

has two stable models, $\{p\}$ *and* $\{q\}$*. The programs*

$$\Pi_4 = \{p :- \ not\ p\} \ \ and \ \ \Pi_5 = \{p. \quad :- \ p.\}$$

have no stable models.

We will now present the grounding of arbitrary rules. Let Π_g be the result of grounding all the global variables from Π. If the program is not simple then we remain with local variables in constraint literals of the rules. Let us define the grounding of these local variables.

The program Π can be divided into two parts. Π_d consists of all rules which are definitions of the domain predicates. This is the *domain part* of Π. Π_r consists of the rest of the rules of Π. From the definition of domain predicates, Π_d has no rules with negative recursion. Therefore, Π_d is a stratified program and has exactly one stable model.

Let A be the stable model of the domain part of Π_g. An extension of a domain predicate p in Π is defined as the set of all ground atoms formed by p which belongs to A. From condition (2) of restrictions in smodels system, we know that for any conditional literal $l : d$, every local variable in l appears also in d. An *instantiation* of a conditional literal $l : d$, is $l' : d'$, where $d' \in A$ and l' is formed by replacing all variables in l by its corresponding terms in d'. The grounding of conditional literal $l : d$ is the set of $l's$ such that for some d', $l' : d'$ is an instantiation of $l : d$. That is, given a conditional literal L,

$$p(X) : q(X)$$

if the extension of q is $\{q(a_1), \ldots, q(a_n)\}$ then ground of L is $p(a_1), \ldots, p(a_n)$.

Let the result of grounding all local variables from Π_g be Π'.

Example 2.5 *To illustrate the construction of* Π'*, let us consider a program* Π *:*

$$q(1).$$

$$q(2).$$

$$r(a).$$

$$r(b).$$

$$s :- 1\{p(X,Y) : q(X)\}, r(Y).$$

Π_g *is obtained from grounding the global variable* Y*.*

$$q(1).$$

$$q(2).$$

$$r(a).$$

$$r(b).$$

$$s :- 1\{p(X,a) : q(X)\}, r(a).$$

$$s :- 1\{p(X,b) : q(X)\}, r(b).$$

Π_d *consists of the facts and the extension of* q *is* { *q(1), q(2)* }*. Then the local variable*

X is grounded to give Π'

$$q(1).$$

$$q(2).$$

$$r(a).$$

$$r(b).$$

$$s :- \ 1\{p(1,a), p(2,a)\}, r(a).$$

$$s :- \ 1\{p(1,b), p(2,b)\}, r(b).$$

Even though, the resulting program Π' is ground, the grounding process does not stop here. We will continue the simplification of Π' to a program consisting of simple rules or rules of two special types defined as follows.

A *choice rule* is a statement of the form:

$$\{h_1, \ldots, h_k\} :- \ l_1, \ldots, l_n. \tag{2.7}$$

where h's are atoms and $l_1, \ldots, l_n$ are simple literals.

A *cardinality rule* is a statement of the form:

$$h :- \ k\{l_1, \ldots, l_n\}. \tag{2.8}$$

where h is an atom, $l_1, \ldots, l_n$ are simple literals and k is the lower bound of the only constraint literal in the body. Choice rules and cardinality rules are special types of cardinality constraint rules.

By construction, rules of Π' contains only simple and constraint literals. We simplify the rules which have constraint literals either in the head or the body. A constraint literal in the head of the rule is simplified differently from a constraint literal in the body of the rule. A constraint literal in the head of a cardinality constraint rule,

$$lower\{h_1, \ldots, h_n\}upper :- \ body.$$

is replaced by a choice rule and two cardinality constraint rules with empty head as follows :

$$\{h_1, \ldots, h_n\} :- \ body.$$

$$:- \ upper + 1\{h_1, \ldots, h_n\}, body.$$

$$:- \ n - lower + 1\{not\ h_1, \ldots, not\ h_n\}, body.$$

Example 2.6 *Consider a program* Π' *:*

$$1\{\ p(a), p(b), p(c)\ \}2 :- \ r(d).$$

It is simplified to give,

$$\{\ p(a), p(b), p(c)\ \} :- \ r(d).$$

$$:- \ 3\{\ p(a), p(b), p(c)\ \}, r(d).$$

$$:- \ 3\{\ not\ p(a), not\ p(b), not\ p(c)\ \}, r(d).$$

A constraint literal in the body of a rule,

$$h :- \ lower\{d_1, \ldots, d_n\}upper.$$

is replaced by two simple literals, and two cardinality rules are added to the program as follows :

$$h :- \ int1, not \ int2.$$

$$int1 :- \ lower\{d_1, \ldots, d_n\}.$$

$$int2 :- \ upper + 1\{d_1, \ldots, d_n\}.$$

Here $int1$ and $int2$ are new predicates not in the signature of Π. Therefore, we get a simple rule and two cardinality rules. In example (2.6), the constraint literals in the body are simplifed to give :

$$\{ \ p(a), p(b), p(c) \ \} :- \ r(d).$$

$$:- \ int1, r(d).$$

$$int1 :- \ 3\{ \ p(a), p(b), p(c) \ \}.$$

$$:- \ int2, r(d).$$

$$int2 :- \ 3\{ \ not \ p(a), not \ p(b), not \ p(c) \ \}.$$

Example 2.7 *Consider a program* Π,

$$q(a).$$

$$q(b).$$

$$p(c).$$

$$p(d).$$

$$2\{h(X,Y) : q(X)\}3 :- \ 1\{r(Z) : q(Z)\}1, p(Y).$$

The extension of q *and* p *are* $\{q(a), q(b)\}$ *and* $\{p(c), p(d)\}$, *respectively. The variables* X *and* Z *are local and* Y *is global. We ground the global variables to get :*

$$q(a).$$

$$q(b).$$

$$p(c).$$

$$p(d).$$

$$2\{h(X,c) : q(X)\}3 :- \ 1\{r(Z) : q(Z)\}1, p(c).$$

$$2\{h(X,d) : q(X)\}3 :- \ 1\{r(Z) : q(Z)\}1, p(d).$$

Now the local variables of conditional literals are grounded as :

$$q(a).$$

$$q(b).$$

$$p(c).$$

$$p(d).$$

$$2\{h(a,c), h(b,c)\}3 :- 1\{r(a), r(b)\}1, p(c).$$

$$2\{h(a,d), h(b,d)\}3 :- 1\{r(a), r(b)\}1, p(d).$$

The transformation of the constraint literal in the body is done as :

$$q(a).$$

$$q(b).$$

$$p(c).$$

$$p(d).$$

$$2\{h(a,c), h(b,c)\}3 :- int1, not\ int2, p(c).$$

$$2\{h(a,d), h(b,d)\}3 :- int1, not\ int2, p(d).$$

$$int1 :- 1\{r(a), r(b)\}.$$

$$int2 :- 2\{r(a), r(b)\}.$$

The transformation of the constraint literal in the head is done as :

$q(a).$

$q(b).$

$p(c).$

$p(d).$

$\{h(a,c), h(b,c)\} :- \ int1, not\ int2, p(c).$

$:- \ 4\{h(a,c), h(b,c)\}, int1, not\ int2, p(c).$

$:- \ 1\{not\ h(a,c), not\ h(b,c)\}, int1, not\ int2, p(c).$

$\{h(a,d), h(b,d)\} :- \ int1, not\ int2, p(d).$

$:- \ 4\{h(a,d), h(b,d)\}, int1, not\ int2, p(d).$

$:- \ 1\{not\ h(a,d), not\ h(b,d)\}, int1, not\ int2, p(d).$

$int1 :- \ 1\{r(a), r(b)\}.$

$int2 :- \ 2\{r(a), r(b)\}.$

Finally, all rules are transformed to simple, choice or cardinality rules.

$$\{h(a,c), h(b,c)\} :- \ int1, not\ int2, p(c).$$

$$:- \ int3, int1, not\ int2, p(c).$$

$$int3 :- \ 4\{h(a,c), h(b,c)\}.$$

$$:- \ int4, int1, not\ int2, p(c).$$

$$int4 :- \ 1\{not\ h(a,c), not\ h(b,c)\}.$$

$$\{h(a,d), h(b,d)\} :- \ int1, not\ int2, p(d).$$

$$:- \ int5, int1, not\ int2, p(d).$$

$$int5 :- \ 4\{h(a,d), h(b,d)\}.$$

$$:- \ int6, int1, not\ int2, p(d).$$

$$int6 :- \ 1\{not\ h(a,d), not\ h(b,d)\}.$$

$$int1 :- \ 1\{r(a), r(b)\}.$$

$$int2 :- \ 2\{r(a), r(b)\}.$$

We see that, for any program Π, $ground(\Pi)$ consists of only simple, choice and cardinality rules. There are no conditional literals in ground(Π). We now give semantics for ground(Π).

Definition 2.3 *A set of atoms S satisfies a cardinality constraint C of the form* (2.3) *(denoted as $S \models C$) iff $lower \leq W(C,S) \leq upper$ where*

$$W(C,S) = |\{l \in lit(C) : S \models l\}|$$

is the number of literals in C satisfied by S.

The notion of S *satisfies a rule* r and S *satisfies a program* Π is the same as the one for simple programs.

The reduct of a cardinality rule r :

$$h :- \ k\{a_1, \ldots, a_n, not\ b_1, \ldots, not\ b_m\}. \tag{2.9}$$

w.r.t. a set of atoms S is defined as follows. Let C be the constraint literal in the body of r.

$$r^S = \left\{ \ h :- \ k'\{a_1, \ldots, a_n\}. \quad \text{where } k' = k - |\{b_1, \ldots, b_m\} \setminus S| \right.$$

The reduct of a choice rule r :

$$\{h_1, \ldots, h_k\} :- \ a_1, \ldots, a_n, not\ b_1, \ldots, b_m. \tag{2.10}$$

is as follows. If C is the constraint literal in the head of r then,

$$r^S = \begin{cases} \emptyset & \text{if } b_1, \ldots, b_m \cap S \neq \emptyset \\ h_i :- \ a_1, \ldots, a_n. & \forall h_i \in lit(C) \cap S, \ \text{otherwise.} \end{cases}$$

The notion of reduct of Π(2.6), deductive closure of Π(2.1), and definition of a stable model (2.2) of a program are the same as before.

Example 2.8 *Consider a program* Π_0*,*

$$\{ \ a_1, a_2, a_3, a_4 \ \}$$

and $S = \{ \ a_1 \ \}$*, the reduct* Π_0^S *is*

$$a_1$$

and S is a stable model of Π_0*.*

For the program Π_1*,*

$$3\{\ a_1, a_2, a_3, a_4\ \}4$$

the stable models are $\{a_1, a_2, a_3\}$, $\{a_1, a_2, a_4\}$, $\{a_1, a_3, a_4\}$, $\{a_2, a_3, a_4\}$, $\{a_1, a_2, a_3, a_4\}$.

CHAPTER III

ALGORITHM

In this chapter, we present *new_smodels*, the algorithm for computing the stable models of a ground program of $\mathcal{SL}$. The algorithm is a modification of the *smodels* algorithm from [13] described below:

3.1 The *smodels* Algorithm

We will start with introducing some terminology and notation. Let Π be a program of $\mathcal{SL}$ with signature Σ and B be a set of simple literals of Σ.

- An atom a and its negation, *not* a, are called a **complementary pair**. We will identify an expression *not* (*not* a) with a.

- $B^+ = \{a \in Atoms(\Sigma) \mid a \in B\}$,

  $B^- = \{a \in Atoms(\Sigma) \mid not\ a \in B\}$,

  $Atoms(B) = B^+ \cup B^-$.

- A set, S, of atoms is **compatible** with B if for every atom $a \in Atoms(\Sigma)$,

  (1) if $a \in B^+$ then $a \in S$,

  (2) if $a \in B^-$ then $a \notin S$.

- B **covers** a set of atoms S, $covers(B, S)$, if

$$S \subseteq Atoms(B).$$

We consider the following rules throughout the chapter.

A simple rule:

$$h :- \ l_1, \ldots, l_n. \tag{3.1}$$

A choice rule:

$$\{h_1, \ldots, h_m\} :- \ l_1, \ldots, l_n. \tag{3.2}$$

A cardinality rule:

$$h :- \ L\{l_1, \ldots, l_n\}. \tag{3.3}$$

where l's are literals and h's are atoms of a signature Σ and L is the lower bound of the cardinality rule. Given a rule r, $head(r)$ denotes the set of atoms in the head and $body(r)$ denotes the set of literals in its body.

Definition 3.4 (a) Rules (3.1), (3.2) are *falsified* by a set of simple literals B, if there exists a literal $l_i \in body(r)$ such that $not\ l_i \in B$;

(b) Rule (3.3) is *falsified* by B, if $n - |\{not\ l_i : l_i \in \{l_1, \ldots, l_n\}\} \cap B| < L$ i.e., the number of literals in $\{l_1, \ldots, l_n\}$ which are not falsified by B is smaller than L.

Definition 3.5 The **reduced form**, r_B, of r with respect to a set of simple literals B is :

1. If r is a simple rule then

$$r_B = \begin{cases} \emptyset & \text{if } r \text{ is falsified by } B \\ h :- \mathit{body}(r) \setminus B. & \text{otherwise.} \end{cases}$$

2. If r is a choice rule then

$$r_B = \begin{cases} \emptyset & \text{if } r \text{ is falsified by } B \\ \{h_1, \ldots, h_m\} :- \mathit{body}(r) \setminus B. & \text{otherwise.} \end{cases}$$

3. If r is a cardinality rule and C is the constraint literal in the body then

$$r_B = \begin{cases} \emptyset & \text{if } r \text{ is falsified by } B \\ h :- L'\{\ \mathit{lit}(C) \setminus \mathit{Atoms}(B)\ \}. & \mathit{where}\ L' = L - |\mathit{lit}(C) \cap B|. \text{ Otherwise.} \end{cases}$$

Given a program Π and a set of literals B, the **reduced form** of Π with respect to B, $r(\Pi, B)$ is defined as

$$\{\ r_B \mid r \in \Pi\ \}.$$

$r(\Pi, B)$ is the set of all **active** rules in Π w.r.t. B.

3.1.1 The Main Computation Cycle

The function **smodels** forms the main loop of the computation process as shown in Figure 3.1. The inputs for the function are a ground program Π, a set of literals B, a boolean $found$. If $found$ is true then the function returns a stable model of Π compatible with B. Otherwise, there is no such stable model. It uses function **expand** which computes the set of conclusions derivable from a set S of literals using the rules of Π, function **pick** selecting a literal undecided by S, and a self-explanatory funcion **backtrack**. The accurate description of these functions will be given in the following sections.

The algorithm performs the following steps:

1. Initializes S to empty, $found$ to true and Y to the *lower closure* of Π with respect to $\emptyset$. The computation of lower closure is discussed in section 3.1.2.

2. Procedure **expand** computes the set C of consequences of Π and $B \cup Y$. If C is consistent then it stores C in S. Otherwise, $conflict$ is set to true and S is unchanged. The computation of these consequences is defined by the closure rules described in section 3.1.2. The set of literals stored in S after this call to *expand* has the following properties :

    - $B \subseteq S$
    - every stable model that is compatible with B is compatible with S.

3. If $conflict$ is true then there is no stable model of Π compatible with B and $found$ is set to false.

4. If $found$ is true then the loop containing the steps (a)-(d) below is executed. The loop terminates if there is no stable model of Π compatible with B (in this

function $smodels(\Pi : program, B : set\_of\_lits, \mathbf{var} found : bool) : set\_of\_atoms$

% **postcondition :** if there is no stable model of Π compatible with B then

% found is set to false. Otherwise found is set to true and *smodels* returns a

% stable model of Π that is compatible with B.

VAR S : stack of literals; **VAR** Y : set of literals; **VAR** conflict : bool;

initialize(S); Y := $lc(\Pi, \emptyset)$; found := true;

expand(Π, S, $B \cup Y$, conflict);

if $conflict$ **then** $found := false$;

while $not\ covers(S, Atoms(\Pi))$ **and** $found$ **do**

$pick(l, \overline{S})$;

expand(Π, S, $\{l\}$, conflict);

if $conflict$ **then** expand(Π, S, $\{not\ l\}$, conflict);

while $conflict$ **and** $found$ **do**

x := back_track(Π, S, found);

if $found$ **then** expand(Π, S, $\{x\}$, conflict);

return $S \cap Atoms(\Pi)$;

Figure 3.1: **smodels** algorithm - computation of stable models

case found is set to false), or if for each atom $a \in Atoms(\Pi)$, a is defined in S, i.e., either $a \in S$ or *not* $a \in S$.

a. the function **pick** will choose a literal l undefined in S.

b. **expand** computes the set X of consequences of Π and $S \cup \{ \ l \ \}$. If X is consistent, then it is stored in S and the corresponding occurrence of l, is marked as a *picked literal*. Otherwise, S is unchanged and $conflict$ is set to true.

c. If there is no conflict in S then control goes to step (4). Otherwise, **expand** computes the consequences of Π and $S \cup \{ \ not \ l \ \}$ and stores it in S, if they are consistent. If the set of consequences is not consistent then *expand* sets $conflict$ true, and leaves S unchanged.

d. If $conflict$ is true then the steps (i), (ii) of the inner loop are executed.

   (i) The function **backtrack** pops literals from S until it finds a picked literal x, pops it from S, and returns *not* x. If the function doesn't find a picked literal in S then $found$ is set to false.

   (ii) if *found* is false then there is no stable model of Π compatible with B. Otherwise **expand** finds the consequences of Π and $S \cup \{ \ not \ x \ \}$, and stores them in S, if they are consistent, else $conflict$ is set to true and S is unchanged.

5. *smodels* returns atoms in S.

If *found* is true when the outer loop exits, then $S \cap Atoms(\Pi)$ is a stable model of Π that is compatible with B. Otherwise, there are no stable models of Π that are compatible with B.

3.1.2 The expand Cycle

Before we describe the expand procedure, we need to introduce some terminology. A set of simple literals U, is called the **lower closure** of a program Π with respect to a set of simple literals B, if it is a minimal set containing B and closed under the following six inference rules.

1. If r is a simple rule (3.1), and $body(r) \subseteq U$, then $h \in U$.

2. If r is a cardinality rule (3.3), and $|\{l_1 \ldots, l_n\} \cap U| \geq L$, then $h \in U$.

3. If an atom a is not in the head of any rule in Π, then $not\ a \in U$.

4. If r is the only rule of Π such that $h \in head(r)$, $h \in U$ and r is of the type (3.1) or (3.2), then $body(r) \subseteq U$.

5. If h is in the head of a simple rule or a choice rule r, in Π, $not\ h \in U$, and all literals in the body of r except l_i belong to U, then $not\ l_i \in U$.

6. If h is the head of a cardinality rule r (3.3) in Π, and *not* $h \in U$, and $|\{l_1, \ldots, l_n\} \cap U| = L - 1$ then, if $l_i \notin U$ then *not* $l_i \in U$.

This closure will be denoted by $lc(\Pi, B)$. For the sake of easier representation, we have split the four inference rules into the six rules which we have just discussed.

Proposition 3.6 *Let* Π *be a program, and* B *be a set of simple literals. By [14], we have the following:*

1. If S *is a stable model of* Π*, then* B *is compatible with* S *iff* $lc(\Pi, B)$ *is compatible with* S*.*

2. $lc(\Pi, B)$ *is unique.*

The *expand* routine calls a procedure *atleast*, which computes the lower closure of a program and a set of literals.

3.1.2.1 The atleast procedure

The procedure **atleast** (Figure 3.2) takes the program Π, the stack of literals S, and a set of literals X as inputs, and computes the *lower closure* of Π with respect to $S \cup X$. The function lc computes the lower closure of Π with respect to a single literal l.

For each execution of the loop in the procedure, a literal l is selected from X and pushed to the stack S. If S is inconsistent, then variable $conflict$ is set to true

procedure atleast(**var** Π: Program, **var** S: stack of lits,

X: set of lits, **var** conflict : bool)

% **precondition :** • conflict = false, $\Pi = r(\Pi, S)$

% **postcondition :** • If conflict = true then $\nexists$ stable model compatible with S.

% Otherwise $\Pi = r(\Pi, S)$, $S = lc(\Pi, S)$.

VAR X_0 : $set\_of\_literals$;

while *not empty*(X) **and** *not conflict* **do**

select $l \in X$;

$X := X \setminus \{\ l\ \}$;

$push(l, S)$;

$conflict := conflict(S)$

if *not conflict* **then**

$X_0 := lc(\Pi, \{l\})$;

$X := (X \cup X_0) \setminus S$;

$\Pi := r(\Pi, \{l\})$;

end if

end procedure

Figure 3.2: atleast procedure

and the loop terminates, else $lc(\Pi, \{l\})$ finds the *lower closure* of Π with respect to l and stores it in X. The reduced form of Π with respect to $\{l\}$, $r(\Pi, \{l\})$, is computed and is assigned to Π.

If $conflict$ is true then there is no stable model of Π which is compatible to $S \cup X$. If $conflict$ is false then by Proposition 3.6, every stable model of Π compatible with $S \cup X$ is compatible with $lc(\Pi, S \cup X)$, (computed by *atleast* and stored in S).

expand uses a function to compute the atoms that can possibly be true in a stable model of Π compatible with S. This function is called *atmost* and is explained below.

3.1.2.2 The atmost function

Let us introduce some terminology. A program is called a definite program, if all rules of the program are simple and does not contain not-atoms. Let S be a set of literals and Π be a program. By $\alpha(\Pi, S)$ we denote a definite program obtained from Π by

1. Removing all rules in Π which are falsified by S.

2. Removing from the result of step one,

    a. all not-atoms from the bodies of the simple and choice rules.

b. all not-atoms from the bodies of the cardinality rules and decreasing the lower bound L by the number of not-atoms removed.

3. Replacing each choice rule $r(3.2)$, by simple rules $h_i :- l_1, \ldots, l_n$, for each $h_i \in head(r)$ such that $not\ h_i \notin S$.

The **upper closure** of a program Π with respect to S, denoted as $up(\Pi, S)$, is defined as the deductive closure (2.1) of $\alpha(\Pi, S)$.

$up(\Pi, S)$ corresponds to the set of atoms that may belong to any stable model of Π compatible with S. All stable models of Π compatible with S must consist of atoms belonging to $up(\Pi, S)$.

Proposition 3.7 *Let* Π *be a program and* S *be a set of literals. From [14], we have the following:*

1. If Y *is a stable model of* Π *compatible with* S *then* $Y \subseteq up(\Pi, S)$.

2. $up(\Pi, S)$ *is unique.*

The function **atmost**, computes $up(\Pi, S)$, where Π is the original program and S is a set of literals. Function atmost is not shown here as it is the same in both the smodels and new_smodels algorithms [14]

3.1.2.3 The expand procedure

The *expand* procedure (Figure 3.3) computes the closure of a program Π with respect to $S \cup X$. The inputs of the procedure are a program Π, a stack of literals S, a set of literals X and a boolean variable $conflict$. The procedure has the following main steps :

a. **Expand** stores the initial value of X, S and Π to X_0, S_0 and Π_0, respectively.

b. **Atleast** computes the *lower closure* of Π with respect to $S \cup X_0$ and stores it in S. It computes the reduced form of Π with respect to S.

c. **Atmost** returns $up(\Pi_g, S)$, where Π_g is the original program input to *smodels*. Since atoms which do not belong to $up(\Pi_g, S)$ cannot be consequences of Π and S, expand stores the negation of atoms not present in $up(\Pi_g, S)$ in X_0.

d. The steps (b) and (c) are executed either until no new atoms are added to S or S becomes inconsistent.

e. If $conflict$ is true then S is assigned to S_0 and Π to Π_0.

Proposition 3.8 *Let S_0 and S_1 be the input and output value of S in expand respectively and X be the input set of literals to expand. If conflict is false then a stable model Y of Π is compatible with $S_0 \cup X$ iff Y is compatible with S_1. Otherwise, there is no stable model of Π compatible with $S_0 \cup X$.*

procedure expand(**var** Π: Program, **var** S: stack of lits,

X: set of lits, **var** conflict: bool)

% Let s_0 and π_0 be the initial value of S and Π, respectively.

% **precondition :** • $\Pi = r(\Pi, S)$

% **postcondition :** • $\Pi = r(\Pi, S)$ and if $conflict = true$ then $S = s_0$ and

% $\Pi = \pi_0$. Otherwise, $s_0 \cup X \subseteq S$ and any stable model of Π compatible

% with $S_0 \cup X$ is compatible with S.

VAR $X_0, S', S_0 : set\_of\_literals$;

$X_0 := X$; $S_0 := S$; $\Pi_0 := \Pi$; $conflict := false$;

repeat

$S' := S$;

$atleast(\Pi, S, X_0, conflict)$;

$X_0 := \{\ not\ x |\ x \in Atoms(\Pi) \text{ and } x \notin atmost(\Pi, S)\ \}$;

until $S = S'$ **or** $conflict$;

if $conflict$ **then**

$S := S_0$; $\Pi := \Pi_0$;

end procedure

Figure 3.3: expand procedure

3.1.3 The backtrack function

The function takes a program Π, a stack S and a boolean variable $found$ as inputs. The function pops literals from S until it pops a literal which is a *picked literal*. The negation of the literal is returned. If such a literal is not found in S, then it sets $found$ to false. The function then finds the reduced form of Π_g and the new S. Π_g is the original program input to **smodels** function and is global to all functions. The computed $r(\Pi_g, S)$ is stored as Π. The function is shown in Figure 3.4.

The *backtrack* function is called when $conflict$ is true. This implies that *expand* found a conflict when computing the closure of Π with respect to $S \cup X$ where X contains the last picked literal. According to the proposition (3.8), there exists no stable model compatible with $S \cup X$ and therefore all literals which are consequences of the last picked literal are removed from S, and the negation of the picked literal is returned and *smodels* tries to find a stable model which is compatible with the returned literal. If such a picked literal is not found in S then there exists no stable model of the program compatible with B, and therefore $found$ is set to false.

Proposition 3.9 *If backtrack returns with $found$ as false then there is no stable model of Π compatible with B, where B is the input set of literals to smodels.*

function back_track(**var** Π: Prog, **var** S: stack, **var** found : bool) : lit

% **precondition :** • $\Pi_0 = r(\Pi_0, S_0)$

% **postcondition :** • If $S_0 = S_1\ x\ S_2$, where x is a picked literal, and S_2

% contains no picked literals, then $S = S_1$ and $\Pi = r(\Pi, S)$

% Otherwise such a literal does not exist and $found$ is false.

VAR x : *lit*

$x := pop(S);$

while $S \neq \emptyset$ **and** $x \neq picked\_literal$ **do**

$x := pop(S);$

end while

$\Pi := r(\Pi_g, S);$ % Π_g is the original program and is global.

if $S = \emptyset$ **and** $x \neq picked\_literal$ **then**

$found := false;$

return *not* x;

end function

Figure 3.4: backtrack function

3.1.4 The pick Function

Function **pick** takes as input the set of atoms, $Y = Atoms(\Pi) \setminus Atoms(S)$. It returns a literal formed from atoms of Y. These picked literals are called choice points. The choice points determine the search space in computing stable models of a program. Efficiency of an implementation depends on the literals picked, i.e., the choice points. The implementation of the function is more complex and involves a heuristic function to find the most desirable literal from Y, to achieve efficiency in computing the stable models. Both *smodels* and *Surya* have almost similar implementations of the heuristic function, which is not discussed in this thesis. To know more about the heuristic function used in *smodels* refer to [14].

3.1.5 The lookAhead Function

There is a function called *lookahead* in both *smodels* and *new_smodels* algorithms, with similar implementations in *smodels* and *Surya*. Given a program Π and a stack of literals S, the main use of this function is to find any literal x which returns $conflict$ true for the call $expand(\Pi, S, \{x\}, conflict)$. The negation *not* x is added to S as there exists no stable model of Π compatible with $S \cup \{x\}$. This prunes the search space for finding a model and returns a model faster. The efficiency increase caused by *lookahead* is considerable. More information about *lookahead* can be found in [14]. This function is not discussed further in this thesis.

3.2 The *new_smodels* Algorithm

Given a program Π with variables, *lparse* returns its grounded form Π_g. The function **new_smodels** takes as input the ground program Π_g, a set of literals B, and a set of *constraint sets* of the program Π (which will be discussed shortly) and returns a stable model of Π compatible with B, if one exists else it reports failure.

Before we discuss the algorithm, we need to know more about grounding of choice rules and cardinality rules by *lparse* and also about *constraint sets*.

3.2.1 Lparse Grounding

lparse starts its work with grounding of the domain part, Π_d, of Π (see section 2.2) and uses the result to compute extensions of Π's domain predicates w.r.t. the stable model of Π_d. (Recall that by extension of predicate p in stable model S we mean the collection of atoms of S formed by p.) These extensions are used to ground the remaining rules of Π. Below we will describe the groundings of two types of rules which play especially important role in *new_smodels* algorithm.

a. Grounding of simple choice rules.

A choice rule r of Π is called a *simple choice rule* if the body of r consists of only literals formed by domain predicates from Σ.

A simple choice rule r is grounded by lparse into a collection of three types of rules, called *ground instances* of r. We will illustrate this notion by the following example.

Example 3.1 *Let us consider the following simple choice rule with a single domain predicate in its body.*

$$L \ \{ \ p(X,Y) \ : \ q(X) \ \} \ U \ \ :- \ \ r(Y). \tag{3.4}$$

Let the extensions of q *and* r *be* $\{q(1), q(2), q(3)\}$ *and* $\{r(a), r(b), r(c)\}$*, respectively. Then, the grounding of (3.4) produces the following rules:*

$$\begin{aligned}
&\{ \ p(1,a), \ \ p(2,a), \ \ p(3,a) \ \}. \\
&\{ \ p(1,b), \ \ p(2,b), \ \ p(3,b) \ \}. \\
&\{ \ p(1,c), \ \ p(2,c), \ \ p(3,c) \ \}.
\end{aligned} \tag{3.5}$$

$$\begin{aligned}
&:- \ U{+}1 \ \{ \ p(1,a), \ \ p(2,a), \ \ p(3,a) \ \}. \\
&:- \ U{+}1 \ \{ \ p(1,b), \ \ p(2,b), \ \ p(3,b) \ \}. \\
&:- \ U{+}1 \ \{ \ p(1,c), \ \ p(2,c), \ \ p(3,c) \ \}.
\end{aligned} \tag{3.6}$$

$$\begin{aligned}
&:- \ n{-}L{+}1 \ \{ \ not \ \ p(1,a), \ \ not \ \ p(2,a), \ \ not \ \ p(3,a) \ \}. \\
&:- \ n{-}L{+}1 \ \{ \ not \ \ p(1,b), \ \ not \ \ p(2,b), \ \ not \ \ p(3,b) \ \}. \\
&:- \ n{-}L{+}1 \ \{ \ not \ \ p(1,c), \ \ not \ \ p(2,c), \ \ not \ \ p(3,c) \ \}.
\end{aligned} \tag{3.7}$$

where $n = 3$ *is the number of simple literals in the body of the constraint from (3.7).*

(Notice that all such constraints have the same number of simple literals in them.)

The collection of rules of the type (3.6) and (3.7) of a simple choice rule r are referred as $C_p(r)$ and $C_n(r)$, respectively.

b. Grounding of simple cardinality rules.

A cardinality constraint rule (2.4) of the form:

$$:- L\{l_1, \ldots, l_n\}, \Gamma. \tag{3.8}$$

where Γ consists of simple literals formed from domain predicates, is called a *simple cardinality rule*. We show the *ground instances* of a simple cardinality rule by an example.

Example 3.2 *Let us consider the following rule cr with a single domain predicate in its body.*

$$:- L\ \{\ p(X,Y)\ :\ q(X)\ \},\ r(Y). \tag{3.9}$$

Let the extensions of q *and* r *be* $\{q(1), q(2), q(3)\}$ *and* $\{r(a), r(b), r(c)\}$*, respectively. Then, lparse grounds (3.9) as follows,*

$$:- L\ \{\ p(1,a),\ p(2,a),\ p(3,a)\ \}.$$

$$:- L\ \{\ p(1,b),\ p(2,b),\ p(3,b)\ \}.$$

$$:- L\ \{\ p(1,c),\ p(2,c),\ p(3,c)\ \}.$$

The collection of these rules is referred as $C_p(cr)$. Often it will be convenient to identify the C_n for a cardinality rule with the empty set.

3.2.2 Some Definitions

Consider a cardinality rule r of the form,

$$:-\ L_{cr}\ \{\ l_1, \dots,\ l_n\ \}. \tag{3.10}$$

Recall that $lit(r) = \{\ l_1, \dots,\ l_n\ \}$. If C is a collection of cardinality rules $\{r_1, \dots, r_n\}$, then $lit(C) = lit(r_1) \cup \dots \cup lit(r_n)$.

Definition 3.10 *Let C be a collection of cardinality rules. The* reduced form *of C with respect to a set of simple literals S is*

$$r(C, S) = \{\ r_S \mid r \in C\}.$$

where r_S is the reduced form of cardinality rule r with respect to S introduced in definition 3.5. Let $\mathcal{C} = \{C_1, \dots, C_n\}$, where C_i's are sets of cardinality rules. The reduced form *of $\mathcal{C}$ with respect to S is :*

$$r(\mathcal{C}, S) = \{\ r(C_i, S) \mid C_i \in \mathcal{C}\}.$$

Example 3.3 *Given $\mathcal{C} = \{C_1, C_2\}$, where*

$C_1 = \{\ \ :-\ 2\{a, b, c\},\ :-\ 1\{d, e\},\ :-\ 3\{f, g, h\}\ \},$

$C_2 = \{\ \ :-\ 1\{not\ a, not\ b, not\ c\},\ :-\ 3\{not\ d, not\ e\},\ :-\ 1\{not\ f, not\ g, not\ h\}\ \}$

and $S = \{\ a, not\ d, not\ e, f\}$, we get $r(\mathcal{C}, S) = \{C_1', C_2'\}$, where,

$$C_1' = r(C_1, S) = \{\ \ :-\ 1\{b, c\},\ :-\ 2\{g, h\}\ \},$$

$$C_2' = r(C_2, S) = \{\ \ :-\ 1\{not\ b, not\ c\},\ :-\ 1\{not\ g, not\ h\}\ \}.$$

A set of literals S $satisfies$ a set of cardinality rules C, if S satisfies all rules that belong to C.

Proposition 3.11 *Let S be a set of literals and C be a set of cardinality rules. S satisfies C iff S satisfies $r(C, S)$.*

Definition 3.12 *Let C be a collection of cardinality rules $r_1, \ldots, r_k$ with lower bounds $L_1, \ldots, L_k$ and empty heads. By* **merge** *of C we denote the cardinality rule,*

$$merge(C) = \ :- \ L_1 + \ldots + L_k - k + 1\{ \ lit(C) \ \}. \tag{3.11}$$

If $\mathcal{C} = \{C_1, \ldots, C_n\}$ is a collection of sets of cardinality rules then

$$merge(\mathcal{C}) = merge(\{merge(C_1), \ldots, merge(C_n)\}).$$

Example 3.4 *Consider $\mathcal{C}' = \{C'_1, C'_2\}$ from example 3.3, then $merge(\mathcal{C}')$ is*

$$:- \ 2\{b, c, g, h, not\ b, not\ c, not\ g, not\ h\}.$$

where,

$$C_1^M = merge(C'_1) = \ :- \ 2\{b, c, g, h\},$$

$$C_2^M = merge(C'_2) = \ :- \ 1\{not\ b, not\ c, not\ g, not\ h\}.$$

Proposition 3.13 *Let S be a set of literals and C be a set of cardinality rules. S satisfies C iff S satisfies $C \cup merge(C)$.*

Two sets of cardinality rules C_1 and C_2 are said to be **related**, denoted as $related(C_1, C_2)$, if $lit(C_1) = \{not\ l \mid l \in lit(C_2)\}$. Recall that the above equality is the equality of bags and that two bags X and Y of literals are equal if, for every literal l the number of occurrences of l in X is equal to the number of occurrences of l in Y.

Example 3.5 *Consider the set of cardinality rules,*

$$C_1 = \{\ :-\ 2\{a, a, c\},\quad :-\ 1\{e, f, g\}\ \},$$

$$C_2 = \{\ :-\ 2\{not\ a, not\ g, not\ c\},\quad :-\ 1\{not\ a, not\ e, not\ f\}\ \},$$

$$C_3 = \{\ :-\ 2\{not\ a, not\ g, not\ c\},\quad :-\ 1\{not\ e, not\ f\}\ \}.$$

it is easy to see that C_1 and C_2 are related but C_1 and C_3 are not.

Definition 3.14 *Let R be the set of all simple cardinality and simple choice rules in Π. The sets $C_p(r)$ and $C_n(r)$ for rules of R will be referred to as* **constraint sets** *of Π. The* **collection of constraint sets** *of* Π *is :*

$$\mathcal{CS} = \{C_p(r) \mid r \in R\}\ \cup\ \{C_n(r) \mid r \in R\}.$$

$\mathcal{CS}$ will serve as an input to *new_smodels*.

3.2.3 The Main Computation Cycle

The **new_smodels** algorithm presented in Figure 3.5, is similar to the **smodels** algorithm. Given a program Π, a set of literals B and the collection of constraint

```
function  new_smodels(Π : program, B : set_of_lits, CS : set of C_Sets,
                                        var found : bool ) : set_of_atoms
  VAR S : stack of literals; VAR Y : set of literals; VAR conflict : boolean;
  VAR cp, lb : array [C ∈ CS, C' ∈ CS] of integer;
  VAR nc : array [a ∈ Atoms(Π), C ∈ CS] of integer;
   initialize(S, cp, nc, lb);   Y := lc(Π, ∅);   found := true;
   new_expand(Π, S, B ∪ Y, cp, nc, lb, conflict);
   if conflict then found := false;
   while not covers(S, Atoms(Π)) and found do
      pick(l, S̄);
      new_expand(Π, S, {l}, cp, nc, lb, conflict);
      if conflict then new_expand(Π, S, {not l}, cp, nc, lb, conflict);
      while conflict and found do
         x := new_backtrack(Π, S, cp, nc, lb, found);
          if found then new_expand(Π, S, {x}, cp, nc, lb, conflict);
   return S ∩ atoms(Π);
```

Figure 3.5: **new_smodels** algorithm - computation of stable models

sets $\mathcal{CS}$ of Π, **new_smodels** returns the stable model of Π compatible with B, if one exists. Otherwise, it returns failure.

new_smodels uses routines *new_expand* and *new_backtrack* similar to routines *expand* and *back_track*. Both, *expand* and *new_expand* return $conflict$ when they discover that S cannot be expanded to the desired model of Π. The difference is that *new_expand* merges constraint sets in $\mathcal{CS}$, and as a result, finds conflict substantially faster than *smodels* in some cases. All the other routines used in the main computation cycle are the same as in *smodels*.

To compute the merge of the constraint sets in $\mathcal{CS}$ efficiently, *new_smodels* uses three arrays called *cp*, *nc* and *lb*. We need to know more about the arrays before we describe the main algorithm.

3.2.3.1 The book_keeping tables

Let S, a set of literals from Σ_Π, be a candidate model of Π. Let $\mathcal{CS}$ be the set of constraint sets of the program Π. Unless otherwise specified, we only talk about constraint sets that belong to $\mathcal{CS}$.

From now onwards, we denote $merge(r(C_1, S), r(C_2, S))$ as $mr(C_1, C_2, S)$. *cp* and *lb* are tables indexed by constraint sets of $\mathcal{CS}$. *nc* is a table indexed by atoms occuring in $\mathcal{CS}$, and constraint sets of $\mathcal{CS}$. Let C_1 and C_2 be constraint sets in $\mathcal{CS}$.

1. If $related(C_1, C_2)$, then $cp[C_1, C_2]$ contains the number of complementary pairs

in $mr(C_1, C_2, S)$ else $cp[C_1, C_2] = -1$.

2. If $related(C_1, C_2)$, then $lb[C_1, C_2]$ contains the lower bound of $mr(C_1, C_2, S)$ else $lb[C_1, C_2] = -1$.

3. If $a \notin C_1$ then $nc[a, C_1] = -1$. Otherwise, if atom a is undefined in S then $nc[a, C_1]$ contains the number of occurrences of a in $r(C_1, S)$. If a is defined in S, then the value of $nc[a, C_1]$ holds the value of $nc[a, C_1]$ when a was last undefined.

In what follows, we will denote conditions (1), (2) and (3) above by $\upsilon(cp, S)$, $\upsilon(lb, S)$, and $\upsilon(nc, S)$, respectively. From examples 3.3 and 3.4, $cp[C_1, C_2] = 4$, $lb[C_1, C_2] = 2$ and for $x \in \{b, c, g, h\}$, $nc[x, C_1] = 1$, and $nc[x, C_2] = 1$. Note that in $C = \,:-\, 1\{a, a, not\ c\}$, the number of occurrences of atom a is two and the number of occurrences of atom c is one.

Now, let us go to the algorithm in Figure 3.5. The **precondition** for the algorithm is that $\mathcal{CS}$ *is the set of constraint sets of the program* Π and the **postcondition** is that *if there is no stable model of* Π *compatible with* B *then found is set to false and the return value is undefined. Otherwise, found is set to true and new_smodels returns a stable model of* Π *that is compatible with* B. The algorithm performs the following steps :

1. The procedure **initialize(S,cp, nc, lb)**, initializes the stack S to $\emptyset$. For every

related C_1 and C_2, $cp[C_1, C_2]$ is initialized to $|lit(C_1)|$ and $lb[C_1, C_2]$ is initialized to $\sum_{r \in C_1} L_r + \sum_{r \in C_2} L_r - |C_1| - |C_2| + 1$ and if $a \in Atoms(lit(C_1))$ then $nc[a, C_1]$ is the number of occurrences of a in $lit(C_1)$. It is easy to check that this assignment satisfies the conditions $v(cp, S)$, $v(nc, S)$ and $v(lb, S)$. The *lower closure* of Π with respect to S is computed and stored in Y.

2. The function **new_expand** computes the set Q of consequences of Π and $B \cup Y$. If Q is consistent then it stores Q in S. Otherwise, $conflict$ is set to true and S is left unchanged.

   In its computation *new_expand* uses the closure rules defined in section 3.1.2 together with operation *merge* on constraint sets of $\mathcal{CS}$. The latter allows to derive falsity if S does not satisfy $merge(C)$, where $C \in \mathcal{CS}$. If S is changed during the computation, then the function updates the values of the three arrays to maintain conditions $v(cp, S)$, $v(nc, S)$ and $v(lb, S)$. The arrays are updated using procedures $update1$ and $update2$, which will be discussed in section 3.2.4. The set of literals stored in S after the execution of *new_expand* has the following properties :

   - $B \subseteq S$
   - every stable model that is compatible with B is compatible with S.

3. If $conflict$ is true then $found$ is set to false and there is no stable model of Π

compatible with B.

4. If S does not cover all atoms in Π and $found$ is true then the loop containing steps (a)-(d) below is executed.

   a. function **pick** selects a literal l undefined in S.

   b. **new_expand** finds the consequences of Π and $S \cup \{l\}$. If the consequences are consistent then they are stored in S and the arrays cp, lb, nc are updated to maintain the conditions, $\upsilon(cp, S)$, $\upsilon(nc, S)$ and $\upsilon(lb, S)$. The literal l is marked as a picked literal. Otherwise, S and the arrays are unchanged and $conflict$ is set to true.

   c. If $conflict$ is false then control goes to step (5). Otherwise, **new_expand** finds the consequences of Π and $S \cup \{not\, l\}$. If the consequences are consistent with S then they are stored in S else $conflict$ is set to true. The arrays are updated accordingly.

   d. If there is $conflict$ then steps (i) and (ii) below are performed repeatedly until there is no $conflict$ or until $found$ is false, that is, there is no stable model of Π compatible with B.

      (i) function **new_backtrack** removes literals from S until it finds a picked literal x. The negation of the literal, $not\, x$ is returned. The arrays are updated with respect to the new S by two functions called *back_update*1 and *back_update*2. They will be discussed in section 3.2.5. If the function doesn't find a picked literal then $found$ is set to false.

(ii) If $found$ is true then **new_expand** finds the consequences of Π and $S \cup \{not\ x\}$ and if consistent stores them in S and updates the arrays. Otherwise S and the arrays are unchanged.

6. **new_smodels** returns the atoms in S.

If $found$ is true then the set of atoms returned by *new_smodels* is a stable model of Π compatible with B. Otherwise, there is no such stable model.

3.2.4 The new_expand cycle

The inputs of new_expand are a program Π, a stack of literals S, a set of literals X, a boolean variable $conflict$ and three arrays *cp*, *nc* and *lb*. Intuitively, the new_expand procedure computes the closure of Π and $S \cup X$ the same way as expand does. We begin by explaining all the routines in *new_expand* which are different from routines in *expand*.

The book_keeping tables need to be updated every time *new_expand* adds new literals to S. In doing so, *new_expand* must maintain the conditions $v(cp, S)$, $v(nc, S)$, $v(lb, S)$. Suppose a literal l is added to S, the following are the cases when the arrays change. Let $a = Atoms(l)$,

1. for any $C, C' \in CS$ and $related(C, C')$, if a occurs in C, C' then $cp[C, C']$ and $lb[C, C']$ change. Since a is now defined in S, the value of $nc[a, C]$ for any $C \in CS$ is left unchanged. (This value will be used later by the backtracking procedure.)

2. if there is a rule $r \in C$, such that r is falsified by S because of adding l then for every undefined atom x occuring in r, $nc[x, C]$ changes and hence $cp[C, C']$ and $lb[C, C']$ change.

The following example will help to clarify the above cases when the tables need to be updated.

Example 3.6 *Let us consider the following six rules of a program :*

$$
\begin{aligned}
r_1 &\quad :- \ 4\{a, b, c, d, e, f\}.\\
r_2 &\quad :- \ 4\{g, h, i, j, k, l\}.\\
r_3 &\quad :- \ 4\{m, n, o, p, q, r\}.\\
r_4 &\quad :- \ 3\{not\ a, not\ b, not\ g, not\ h, not\ m, not\ n\}.\\
r_5 &\quad :- \ 3\{not\ c, not\ d, not\ i, not\ j, not\ o, not\ p\}.\\
r_6 &\quad :- \ 3\{not\ e, not\ f, not\ k, not\ l, not\ q, not\ r\}.
\end{aligned}
$$

Let $C = \{r_1, r_2, r_3\}$ *and* $C' = \{r_4, r_5, r_6\}$*. C and C' are related. Let the initial value of* S *be* $S_0 = \{c, e, m, not\ g, not\ l\}$*. By definition, we have* $r(C, S_0) = \{r_{1S_0}, r_{2S_0}, r_{3S_0}\}$

and $r(C', S_0) = \{r_{4S_0}, r_{5S_0}, r_{6S_0}\}$*, where the reduced form of the rules with respect to* S_0 *are:*

$$
\begin{aligned}
r_{1S_0} \quad &:- \ 2\{a, b, d, f\}. \\
r_{2S_0} \quad &:- \ 4\{h, i, j, k\}. \\
r_{3S_0} \quad &:- \ 3\{n, o, p, q, r\}. \\
r_{4S_0} \quad &:- \ 2\{not\ a, not\ b, not\ h, not\ n\}. \\
r_{5S_0} \quad &:- \ 3\{not\ d, not\ i, not\ j, not\ o, not\ p\}. \\
r_{6S_0} \quad &:- \ 2\{not\ f, not\ k, not\ q, not\ r\}.
\end{aligned}
$$

The cardinality rule $M = merge(r(C, S_0), r(C', S_0))$ *is:*

$$
\begin{aligned}
:- \ 11 \ \{ \quad & a, b, d, f, h, i, j, k, n, o, p, q, r, not\ a, not\ b, not\ h, not\ n, \\
& not\ d, not\ i, not\ j, not\ o, not\ p, not\ f, not\ k, not\ q, not\ r \ \}.
\end{aligned}
$$

By definition of the tables, $cp[C, C'] = 13$ *and* $lb[C, C'] = 11$*. The value of* $nc[x, C]$ *is one if* $x \in \{a, b, d, f, h, i, j, k, n, o, p, q, r\}$*. (Note that, the atoms in this example occur only once in* C *or* C'*. In general, an atom "a" can occur more than once in any* $C \in CS$*, and* $nc[a, C]$ *will be equal to the number of such occurrences.)*

Now suppose a new literal "not i", is added to S*, i.e.,* $S_1 = S_0 \cup \{not\ i\}$*. The* *rule r_2 is falsified by S_1 and the new reduced forms of the rules are :*

$$\begin{aligned}
r_{1S_1} &= r_{1S_0}, \\
r_{2S_1} &= \emptyset, \\
r_{3S_1} &= r_{3S_0}, \\
r_{4S_1} &= r_{4S_0}, \\
r_{5S_1} &= \ :-\ 2\{not\ d, not\ j, not\ o, not\ p\}, \\
r_{6S_1} &= r_{6S_0},
\end{aligned}$$

$r(C, S_1) = \{r_{1S_1}, r_{3S_1}\}$, $r(C', S_1) = \{r_{4S_1}, r_{5S_1}, r_{6S_1}\}$,

and $M_1 = merge(r(C, S_1), r(C', S_1))$ *is :*

$$\begin{aligned}
:-\ 7\ \{\ & a, b, d, f, n, o, p, q, r, not\ a, not\ b, not\ h, not\ n, \\
& not\ d, not\ j, not\ o, not\ p, not\ f, not\ k, not\ q, not\ r\ \}.
\end{aligned}$$

To maintain $\upsilon(cp, S_1)$, $\upsilon(nc, S_1)$, $\upsilon(lb, S_1)$, we have to update the values in the arrays as $cp[C, C'] = 9$ and $lb[C, C'] = 7$.

One way to update the bookkeeping tables is to find $r(C, S)$ for all $C \in CS$, and reevaluate the values for the arrays after merge. But it will be more efficient to do the update using the previous values of the tables and the knowledge of the literals added to S. This is done by two procedures *update*1 and *update*2. Let us now see how the values in the arrays are updated by these procedures.

procedure update1(**var** cp, nc : tables, **var** lb : table, S : stack_of_lits)

% **precondition :** • Let Y be $S \setminus \{top(S)\}$. $\upsilon(cp, Y)$, $\upsilon(nc, Y)$, $\upsilon(lb, Y)$.

% **postcondition :** Let a = Atoms(top(S)), then

% • $\forall C, C' \in CS$, if related(C,C') and $a \in lit(C)$ then cp[C,C']= # comp pairs

% in $mr(C, C', Y)$ − # of comp pairs containing $''a''$ in $mr(C, C', Y)$; and

% • lb[C,C'] = lower bound of $mr(C, C', Y)$ - $\alpha(top(S), C, C', Y)$.

VAR a : atom;

a := Atoms(top(S));

for each $C \in CS$ such that $nc[a, C] \geq 0$ **do**

for each $C' \in CS$ such that $related(C, C')$ **do**

$cp[C, C'] := cp[C, C'] - min(nc[a, C], nc[a, C'])$

if $a = top(S)$ **then** % lit(C) consists of atoms.

$lb[C, C'] := lb[C, C'] - nc[a, C]$

else

$lb[C, C'] := lb[C, C'] - nc[a, C']$

end procedure

Figure 3.6: update1 procedure

3.2.4.1 The update1 procedure

The procedure update1, illustrated in Figure 3.6, is one of the two procedures used to update the arrays cp, nc and lb. The precondition for the procedure is that the three arrays satisfy the conditions $v(cp, Y)$, $v(nc, Y)$ and $v(lb, Y)$, where Y is $S \setminus top(S)$ and $top(S)$ is the top element of S. To introduce the post conditions, we need some notations. Let us assume that $lit(C)$ consists of atoms and $lit(C')$ consists of not-atoms for $C, C' \in \mathcal{CS}$. We define $\alpha(top(S), C, C', Y)$, as:

a. The number of occurrences of $l = top(S)$, in $r(C, Y)$, if l is an atom;

b. The number of occurrences of l in $r(C', Y)$, is l is a not-atom.

The postconditions of the procedure are, for all C, C' in CS and $a = Atoms(top(S))$,

1. $cp[C, C']$ gives the number of complementary pairs in $mr(C, C', Y)$ minus the number of occurrences of complementary pair $\{a, not\ a\}$ in $mr(C, C', Y)$, where $mr(C, C', Y) = merge(r(C, Y), r(C', Y))$.

2. $lb[C, C']$ contains the lower bound of $mr(C, C', Y)$ minus $\alpha(top(S), C, C', Y)$.

To see that $update1$ satisfies the corresponding postconditions, let us recall that by $v(cp, Y)$, we mean that for any $C, C' \in CS$, if $related(C, C')$ then $cp[C, C']$ gives the number of complementary pairs in the rule $M = mr(C, C', Y)$. M may contain the occurrences of complementary pair $\{a, not\ a\}$. Suppose $r(C, S)$ contains m occurrences of a and $r(C', S)$ contains n occurrences of $not\ a$, then the number of occurrences of $\{a, not\ a\}$ complementary pair in M is the minimum of m and n. Recall that, $\upsilon(nc, Y)$ says that for any atom $a \in C$ where $C \in CS$, $nc[a, C]$ gives the number of occurrences of a in $r(C, Y)$.

For every related pair $C, C' \in CS$, procedure update1 subtracts the number of complementary pairs formed by atom a in M. This satisfies the first postcondition. For what concerns the second postcondition, it can be seen from the algorithm that, if $top(S)$ is an atom then update1 subtracts from $lb[C, C']$ the number of occurrences of a in C, else it subtracts the number of occurrences of $not\ a$ in C'. Thus, the post conditions are satisfied.

Example 3.7 *Let us consider the rules from example 3.6, and let the value of S be S_0. Conditions $\upsilon(cp, S_0)$, $\upsilon(nc, S_0)$ and $\upsilon(lb, S_0)$ hold, therefore $cp[C, C'] = 13$ and $lb[C, C'] = 11$. Now, let the new value of S be $S_1 = S_0 \cup \{not\ i\}$. We know that $nc[i, C] = 1$, $nc[i, C'] = 1$, $cp[C, C'] = 13$ and $lb[C, C'] = 11$. The values of $cp[C, C']$ and $lb[C, C']$ are updated as follows: Procedure update1 decreases the value of $cp[C, C']$ to 12. $lb[C, C']$ is decreased to 10, as $''not\ i''$ is added to S and $nc[i, C'] = 1$.*

3.2.4.2 The update2 procedure

The **update2** procedure is the second procedure that is used to update the three arrays. The procedure takes as inputs the arrays, a set of rules R and the stack S. The preconditions of the procedure are the postconditions of $update1$, and R is the

procedure update2(**var** cp, nc, lb: tables, R : set_of_rules, S : stack of lits)

% **precondition :** • Let Y be $S \setminus \{top(S)\}$ and $a = Atoms(top(S))$, $\upsilon(nc, Y)$

% • $\forall C, C' \in CS$. if related(C,C') and $a \in lit(C)$ then cp[C,C']= # comp pairs

% in $mr(C, C', Y) -$ # of comp pairs containing $'a'$ in $mr(C, C', Y)$, and

% • lb[C,C'] = lower bound of $mr(C, C', Y)$ - $\alpha(top(S), C, C', Y)$.

% • R is the set of rules inactive w.r.t. S and active w.r.t. Y.

% **postcondition :** • $\upsilon(cp, S)$, $\upsilon(nc, S)$, $\upsilon(lb, S)$

for each $r \in R$ **do**

  if $r \in C$ for some $C \in CS$ **then**

    for every atom x undecided w.r.t. S s.t. x occurs in r **do**

      for every occurrence of x in r **do**

        $nc[x, C] := nc[x, C] - 1$

        for each $C' \in CS$ such that $related(C, C')$ **do**

          if $nc[x, C] < nc[x, C']$ **then** $cp[C, C'] := cp[C, C'] - 1$;

    for each $C' \in CS$ and $related(C, C')$ **do**

      $lb[C, C'] := lb[C, C'] - L_r + 1$

end procedure

Figure 3.7: update2 procedure

set of cardinality rules which are active (not falsified) with respect to $Y = S \setminus top(S)$ but are falsified by S. The postconditions of the procedure are $v(cp, S)$, $v(nc, S)$ and $v(lb, S)$. Let us see how the postconditions are achieved by the procedure.

Since $v(nc, Y)$ is a precondition, $nc[a, C]$ contains the number of occurrences of a in $r(C, Y)$ if $a \in C$. Suppose a rule $r \in R$ belongs to the constraint set $C \in CS$ (note that by definition of constraint sets r can belong to only one $C \in CS$), by definition of $reduced\ form$, $r_S = \emptyset$ and $r_Y \neq \emptyset$. The body of the rule r_Y contains all literals of r undefined in Y and by definition of $r(C, Y)$, $r_Y \in r(C, Y)$. Since the rule r_Y is not present in $r(C, S)$, the number of occurrences of the literals of r in $r(C, S)$, undecided in S, changes. We need to update these changes. For each occurrence of a literal l in r_Y, undecided in S, $nc[a, C]$ decreases by one, where a is the corresponding atom of literal l. This is done in the procedure. Further more, when the number of occurrences of a literal l in $r(C, S)$ is different from that of $r(C, Y)$, the number of complementary pairs in $mr(C, C', S)$ can also change. The number of occurrences of $\{a, not\ a\}$ complementary pairs in $mr(C, C', S)$ is equal to the minimum of $nc[a, C]$ and $nc[a, C']$. The procedure checks and updates this information. From definition of merge and the fact that $Y = S \setminus top(S)$, we get that the lower bound of $mr(C, C', S)$ is equal to the lower bound of $mr(C, C', Y)$ minus the lower bound of r_Y plus one. For every rule $r \in R$, the array lb is updated accordingly. Thus, the postcondition is satisfied.

Example 3.8 *Let us continue with example 3.7. Recall that we have* $cp[C, C'] = 12$*,* $lb[C, C'] = 10$ *and* $S_1 = \{c, e, m, not\ g, not\ l, not\ i\}$*. We have* $R = \{r_2\}$*, and* r_{2S_0} *is* $:- \ 4\{h, i, j, k\}$ *where* $r_2 \in C$*. The atoms undecided in* r_{2S_0} *with respect to* S_1 *are* $\{h, j, k\}$*. Each of these atoms occur only once in* r_{2S_0}*, hence* $nc[h, C] = 1$*,* $nc[j, C] = 1$*,* $nc[k, C] = 1$*. They are decreased in update*2 *by one and become* $nc[h, C] = 0$*,* $nc[j, C] = 0$ *and* $nc[k, C] = 0$*. Since,* $nc[h, C'] = 1$*,* $nc[j, C'] = 1$ *and* $nc[k, C'] = 1$*, we get,* $cp[C, C']$ *is eventually decreased by 3 and* $cp[C, C']$ *becomes 9. The lower bound* $lb[C, C']$ *gets updated to* $10 - 4 + 1$ *that is* 7*, where* 4 *is the lower bound of* r_2*. We see that the values of* $cp[C, C']$ *and* $lb[C, C']$ *are updated to the values calculated in example 3.6.*

3.2.4.3 Function check_constraints

Before we describe the function, let us give the intuition behind EER. The EER consists of two steps:

1. Expand program Π, by adding a new rule $merge(R)$, where R is a set of cardinality rules of Π with same head h. The construction guarantees that, *"A set of literals* S *satisfies* R *iff* S *satisfies* $R \cup merge(R)$*."*

2. Check if the body of $merge(R)$ is satisfied by all stable models of Π containing S. If so then expand S by h.

function check_constraints(Π : program, S : stack, cp, nc, lb : tables) : boolean

% **precondition :** • $\upsilon(cp, S)$, $\upsilon(nc, S)$, $\upsilon(lb, S)$

% **postcondition :** • if *check_constraints* returns **true** then $\nexists$ a stable model

% of Π compatible with S. Otherwise, no decision can be made about the

% existence of a stable model compatible with S.

for each $C, C' \in CS$ s.t. $related(C, C')$ **do**

if $lb[C, C'] \leq cp[C, C']$ **then**

return true

return false

end function

Figure 3.8: check_constraints function

Given two related constraint sets C_1 and C_2 from $\mathcal{CS}$, EER is applied to $R = r(C_1, S) \cup r(C_2, S)$. According to the first part of EER, Π is expanded with a new rule $merge(R)$. In reality, this is not done in the implementation. To apply the inference rule, it suffices to know the number of complementary pairs and lower bound of the cardinality rule $merge(R)$. This is maintained by the two bookkeeping arrays cp and lb. Recall that, for any two related constraint sets C_1, C_2, $cp[C_1, C_2]$ contains the number of complementary pairs in $merge(r(C_1, S), r(C_2, S))$ and $lb[C_1, C_2]$ contains the lower bound of the rule $merge(r(C_1, S), r(C_2, S))$. (Since $r(C_1, S)$ and $r(C_2, S)$ are sets of cardinality rules, we can prove that, $merge(r(C_1, S), r(C_2, S))$ is equal to $merge(r(C_1, S) \cup r(C_2, S))$. That is, $merge(R) = merge(r(C_1, S), r(C_2, S))$.)

To perform the second part of EER, it suffices to compare the number of complementary pairs and lower bound of the merge rule. *For any* $related(C_1, C_2)$*, if* $lb[C_1, C_2] \leq cp[C_1, C_2]$ *then the body of the rule is satisfied for any completion of* S. Therefore, we infer the head of $merge(R)$, which is $false$. Since $false \in S$, there exists no stable model of Π containing S.

Function *check_constraints*, shown in Figure 3.8, takes as input a program Π, a stack of literals S and the three arrays: cp, nc, lb. The preconditions of the function are $\upsilon(cp, S)$, $\upsilon(nc, S)$ and $\upsilon(lb, S)$. The function applies EER to all the related constraint sets in $\mathcal{CS}$. For every $C, C' \in CS$ and $related(C, C')$,the function checks if $lb[C, C'] \leq cp[C, C']$. If there exists $C, C' \in CS$ such that $lb[C, C'] \leq cp[C, C']$ then the function returns true else it returns false.

Proposition 3.15 *If function check_constraints returns true then there exists no stable model of* Π *compatible with* S.

Example 3.9 *Let the inputs of check_constraints be* S_1 *from example 3.8, and* Π*, containing rules* r_1 *to* r_6*, of example 3.6 and* C*,* C' *in* $\mathcal{CS}$ *of* Π*. The conditions on the arrays hold and therefore we have* $cp[C, C'] = 9$ *and* $lb[C, C'] = 7$*, check_constraints* *returns true. By proposition 3.15, there exists no stable model for program* Π*, which is compatible with* S_1*. Furthermore, if* $S = \emptyset$ *then* $cp[C, C'] = 18$ *and* $lb[C, C'] = 16$ *and check_constraints would return true.*

3.2.4.4 The new_atleast procedure

The **new_atleast** procedure is similar to **atleast** procedure of *smodels*. It takes as input a program Π, a stack of literals S, a set of literals X, a boolean $conflict$ and the arrays cp, nc and lb. The preconditions of the procedure are $\upsilon(cp, S_0)$, $\upsilon(nc, S_0)$ and $\upsilon(lb, S_0)$, where S_0 is the input value of S. While X is not empty and $conflict$ is not true, the algorithm loops through the following steps:

1. A literal l is selected from X and pushed on S. Literal l is removed from X.

2. If there is a conflict in S then $conflict$ is set to true.

3. If $conflict$ is false then the steps (a) to (d) below are executed.

    a. Since S is changed by adding a literal l, the values of the tables are updated using **update1**. The preconditions of update1 are satisfied as follows. For the first iteration, the precondition of update1 follows from the precondition of procedure *new_atleast* and for subsequent iterations, the postcondition of procedure update2 acts as precondition of update1. The values of the arrays are updated according to the postcondition of update1.

procedure new_atleast(**var** Π: Program, **var** S: stack of lits, X: set of lits,

var cp, nc, lb: tables, **var** conflict : bool)

% **precondition :** • $\upsilon(cp, S)$, $\upsilon(nc, S)$, $\upsilon(lb, S)$, $conflict = false$

% **postcondition :** • If conflict = false then $\Pi = r(\Pi, S)$, $\upsilon(cp, S)$, $\upsilon(nc, S)$,

% $\upsilon(lb, S)$ and a stable model of Π compatible with $S_0 \cup X$ is compatible with S.

% • Otherwise, $\nexists$ stable model of Π compatible with S.

VAR R : set_of_rules;

while $not\ empty(X)$ **and** $not\ conflict$ **do**

select $l \in X$; $X := X \setminus \{ l \}$; $push(l, S)$;

$conflict := conflict(S)$;

if $not\ conflict$ **then**

$update1(cp, nc, lb, S)$;

$X_0 := lc(\Pi, l)$; $X := (X \cup X_0) \setminus S$; $\Pi := r(\Pi, \{l\}, R)$;

$update2(cp, nc, lb, R, S)$;

if check_constraints(Π, S, cp, nc, lb) **then** $conflict := true$;

end while

end procedure

Figure 3.9: new_atleast procedure

b. The *lower closure* of Π with respect to l is found the same way as in **atleast** (3.1.2) and is added to X. Any literals which are already in S are removed from X, as the lower closure has already been found for these literals. The reduct of the program Π with respect to $\{l\}$ is found and stored in Π. The procedure $r(\Pi, \{l\}, R)$ is different from the one used in *atleast*, here the procedure stores in R all rules of Π falsified by l.

c. **update2** updates the arrays *cp*, *nc* and *lb*. The preconditions of update2 are the postconditions of update1. update2's postconditions are $v(cp, S)$, $v(nc, S)$ and $v(lb, S)$. It uses the set R of rules which are falsified by l. The actual implementation calls *update*2 whenever a rule r that belongs to a constraint set, is falsified. This way R is not actually stored and is used here mainly to simplify the description of algorithm.

d. The function **check_constraints** checks if, for any $related(C, C')$, the lower bound of $mr(C, C', S)$ is less than or equal to the number of complementary pairs in $mr(C, C', S)$. If so then the function returns true and therefore sets $conflict$ to true. The preconditions of the function are $v(cp, S)$, $v(nc, S)$ and $v(lb, S)$. These are the postconditions of *update*2.

Let S_0 and S_1 be the input and output values of S to *new_atleast* respectively. From procedure *new_atleast* and the Propositions 3.6 and 3.15, it follows that :

Proposition 3.16 *a. Procedure new_atleast terminates.*

b. If conflict is true then there exists no stable model of Π *compatible with* S_0*, else every stable model of* Π *compatible with* S_0*, is compatible with* S_1*.*

3.2.4.5 The function atmost

The procedure *atmost* is the same in both the algorithms *new_smodels* and *smodels*. It computes and returns the upper closure of Π with respect to S. The *upper closure* of a program Π with respect to a set of literals S is defined in section 3.1.2.2. The function *atmost* is not shown as they are the same in both the algorithms and can be referred in [14].

3.2.4.6 The new_expand procedure

The **new_expand** procedure is similar to *expand* in *smodels*. The procedure takes as input a program Π, a stack of literals S, a set of literals X, a boolean variable $conflict$ and the three arrays *cp*, *nc* and *lb*. The procedure is illustrated in Figure 3.10. The preconditions are $\Pi_0 = r(\Pi_0, S_0)$, $v(cp, S_0)$, $v(nc, S_0)$ and $v(lb, S_0)$, where S_0 and Π_0 are the input values of Π and S, respectively. Procedure *new_expand* computes the consequences of Π and $S \cup X$ and stores them in S. The following steps are executed in the procedure until, either all the consequences are found, or $conflict$

procedure new_expand(**var** Π: Program, **var** S: stack of lits, X: set of lits,

var cp, nc, lb : tables, **var** conflict : bool)

% **precondition :** • $\Pi_0 = r(\Pi_0, S_0)$, $\upsilon(cp, S_0)$, $\upsilon(nc, S_0)$, $\upsilon(lb, S_0)$

% **postcondition :** • $\Pi = r(\Pi, S)$, $\upsilon(cp, S)$, $\upsilon(nc, S)$, $\upsilon(lb, S)$

% • If $conflict$ is true then $S = S_0$ and $\Pi = \Pi_0$. Otherwise $S_0 \cup X \subseteq S$ and

% any stable model of Π compatible with $S_0 \cup X$ is compatible with S.

VAR $S_t, S', X_t : set\_of\_lits$; **VAR** $\Pi_t : prog$; **VAR** $cp_t, nc_t, lb_t : tables$;

$\Pi_t := \Pi$; $S_t := S$; $X_t := X$; $conflict := false$;

$cp_t := cp$; $lb_t := lb$; $nc_t := nc$;

repeat

$S' := S$;

$new\_atleast(\Pi, S, X_t, cp, nc, lb, conflict)$;

$X_t := \{\ not\ x |\ x \in Atoms(\Pi)$ and $x \notin atmost(\Pi, S)\ \}$;

until $S = S'$ **or** $conflict$;

if $conflict$ **then**

$S := S_t$; $\Pi := \Pi_t$; $cp := cp_t$; $lb := lb_t$; $nc := nc_t$;

end procedure

Figure 3.10: new_expand procedure

is true.

1. The value of S is stored in S'.

2. Procedure **new_atleast** finds the lower closure A of Π with respect to $S \cup X$. If A is consistent and satisfies the constraints sets of $\mathcal{CS}$ then $conflict$ is set to false and A is stored in S. Otherwise $conflict$ is set to true.

3. Function **atmost** finds the upper closure of Π with respect to S, and adds to X the negation of all atoms not in $up(\Pi, S)$.

4. Steps (1) - (3) above are repeated until $S = S'$ or conflict is true.

5. If $conflict$ is true then the values of Π, S and arrays are intialized back to their corresponding input values in *new_expand*. The initial values of Π, S or the arrays are not actually stored but are recomputed by backtracking in the actual implementation. They are shown as being stored here to simplify the presentation of the algorithm.

Proposition 3.17 *(a) Procedure new_expand terminates.*

(b) Let S_0 and S_1 be the input and output value of S in new_expand, respectively. Let X be the input set of literals to new_expand. If conflict is false then a stable model Y of Π is compatible with $S_0 \cup X$ iff Y is compatible with S_1. Otherwise, there is no stable model of Π compatible with $S_0 \cup X$.

3.2.5 The new_backtrack Function

The function **new_backtrack**, illustrated in Figure 3.13, pops literals from S until it finds a picked literal x and returns the negation of x. The *reduced form* of the program Π, with respect to the new set S is computed and stored in Π. If such a literal is not found then *found* is set to false. This part is the same as in the *backtrack* function of *smodels*. In addition, new_backtrack updates the values in the tables *cp*, *nc*, *lb* according to the new S. The procedures *back_update*1 and *back_update*2 are used to update these values.

3.2.5.1 The back_update1 procedure

The *back_update*1 procedure, illustrated in Figure 3.11, performs exactly the reverse of what *update*1 procedure performs. The inputs of the procedure are the book_keeping tables, a literal x and the stack S. The preconditions of the procedure are $v(cp, Y)$, $v(nc, Y)$ and $v(lb, Y)$, where $Y = S \cup \{x\}$. The literal x was removed from S and therefore the values of the arrays have to be updated accordingly.

To understand the computation inside the loop of the procedure, let us suppose the literal $x \in Atoms(lit(C))$ and $x \in Atoms(lit(C'))$, where C and C' are related constraint sets. By definition of *reduced form*, x does not occur in $lit(r(C, Y))$, as $x \in Y$. Since, x is undefined in S, x occurs in $lit(r(C, S))$. Therefore, the number of complementary pairs in $mr(C, C', S) = merge(r(C, S), r(C', S))$, is equal to the

procedure back_update1(**var** cp, nc : tables, **var** lb : tables, x : lit, S : stack_of_lits)

% **precondition :** • $v(cp, Y)$, $v(nc, Y)$, $v(lb, Y)$, where $Y = S \cup \{x\}$

% **postcondition :** $\forall C, C' \in CS$, if related(C,C') and $x \in Atoms(lit(C))$ then

% • cp[C,C'] = # comp pairs in $mr(C, C', Y)$ + # comp pairs of x in $mr(C, C', S)$.

% • lb[C,C'] = lower bound of $mr(C, C', Y) + \alpha(x, C, C', S)$.

VAR a : atom;

$a := Atoms(x)$;

for each $C \in CS$ such that $nc[a, C] \geq 0$ **do**

for each $C' \in CS$ such that $related(C, C')$ **do**

$cp[C, C'] := cp[C, C'] + min(nc[a, C], nc[a, C'])$

if $a = x$ **then** % lit(C) are atoms.

$lb[C, C'] := lb[C, C'] + nc[a, C]$

else

$lb[C, C'] := lb[C, C'] + nc[a, C']$

end procedure

Figure 3.11: back_update1 function

number of complementary pairs in $mr(C, C', Y)$ plus the number of occurrences of complementary pair $\{x, not\ x\}$ in $mr(C, C', S)$. If literal x occurs m times in constraint set $r(C, S)$ and occurs n times in constraint set $r(C', S)$, then the number of occurrences of complementary pair $\{x, not\ x\}$ in $mr(C, C', S)$ would be equal to the minimum of m and n. Since $v(nc, Y)$ is a precondition of $back\_update1$, if atom a is undefined in Y and $a \in lit(C)$, then $nc[a, C]$ is the number of occurrences of a in $r(C, Y)$. We saw that, the value stored in array nc for an atom p after p is defined is not changed. Therefore $nc[p, C]$ contains the number of occurrences of p in C at the time it was defined, i.e., if the stack $S = S_0\ p\ S_1$, where S_0 and S_1 are stack of literals, then $nc[p, C]$ contains the number of occurrences of p in $r(C, S_0)$. Therefore for the literal x, if $a = Atoms(x)$ then the value of $nc[a, C]$ is the number of occurrences of a in $r(C, S)$, and the value of $nc[a, C']$ is the number of occurrences of a in $r(C', S)$. Therefore back_update1 increases $cp[C, C']$ by the minimum of $nc[a, C]$ and $nc[a, C']$. The value $cp[C, C']$, thus computed, will be equal to the number of complementary pairs in $mr(C, C', S)$ if there exists no rule r in C or C', such that, r is not falsified by S, but falsified by Y. If such a rule exists, then $cp[C, C']$ contains an intermediate value which will be further updated in procedure $back\_update2$.

The values of array lb also change for each $C \in CS$ and $x \in Atoms(lit(C))$. Since x is undefined in S, if x is an atom then $lb[C, C']$ increases by the number of occurrences of x in $r(C, S)$. Otherwise, $lb[C, C']$ increases by the number of occurrences of x in $r(C', S)$. The procedure *back_update*1 increases the lower bound accordingly. The value of $lb[C, C']$ will be an intermediate value if there exists a rule $r \in C$ or $r \in C'$, such that, r is falsified by Y but not by S. This is taken care of in procedure *back_update*2.

Example 3.10 *In example 3.8, we saw that the value of* $cp[C, C'] = 9$ *and* $lb[C, C'] = 7$ *for* $mr(C, C', S_1)$. *Let us suppose that* $top(S_1) = not\ i$ *is removed from stack and we need to back_update the values in the arrays using* back_update1. *Since the value of* $nc[i, C] = 1$ *and value of* $nc[i, C'] = 1$, $cp[C, C']$ *is increased by one and is equal to* 10. $lb[C, C'] = 8$ *as "not i" was removed and* $nc[i, C'] = 1$.

3.2.5.2 The back_update2 procedure

The *back_update*2 procedure, illustrated in Figure 3.12, takes as input the tables *cp*, *nc*, *lb*, a set of rules R, a literal x and the stack S. R contains all rules that are falsified in $S \cup \{x\}$ but are active in S. The preconditions of *back_update*2 are the post conditions of *back_update*1. This procedure performs the reverse operations of update2.

To see that *back_update*2 satisfies the corresponding postconditions, note that, all literals undecided in a rule r with respect to $Y = S \cup \{x\}$ are also undefined in S. If $r \in R$ and $r \in C$, then $r_Y = \emptyset$ and $r_S \neq \emptyset$; therefore, the literals undecided in r does not belong to $lit(r(C, Y))$ but belong to $lit(r(C, S))$. These literals increase the

procedure back_update2(**var** : cp, nc, lb : tables, R : set_of_rules,

x : lit, S : stack_of_lits)

% **precondition :** • Let $Y = S \cup \{x\}$, $v(nc, Y)$.

% $\forall C, C' \in CS$, if $related(C, C')$, $M = mr(C, C', Y)$ and $x \in Atoms(lit(C))$ then

% • cp[C,C'] = # comp pairs in M + # comp pairs of x in $mr(C, C', S)$.

% • lb[C,C'] = lower bound of M + $\alpha(x, C, C', S)$.

% • R is the set of rules inactive w.r.t. Y and active w.r.t. S.

% **postcondition :** • $v(cp, S)$, $v(nc, S)$, $v(lb, S)$

for each $r \in R$ **do**

if $r \in C$ for some $C \in CS$ **then**

for every atom a undecided w.r.t. Y s.t. a occurs in r **do**

for every occurrence of a in r **do**

$nc[a, C] := nc[a, C] + 1$

for each $C' \in CS$ such that $related(C, C')$ **do**

if $nc[a, C] \leq nc[a, C']$ **then** $cp[C, C'] := cp[C, C'] + 1$

for each $C' \in CS$ such that $related(C, C')$ **do**

$lb[C, C'] := lb[C, C'] + L_r - 1$

Figure 3.12: back_update2 function

value of $cp[C, C']$ for any C', such that $related(C, C')$. Since $r_S \in r(C, S)$, the lower bound also increases by $L_r - 1$, where L_r is the lower bound of r_S. Arrays cp and $lb$are updated for each occurrence of a literal undecided in r.

Example 3.11 *Consider again example 3.10, where the values of* $cp[C, C'] = 10$ *and* $lb[C, C'] = 8$*. The rule falsified with respect to* S_1 *is* r_2 *from example 3.8. We have* r_2 *as* $:- \ 4\{g, h, i, j, k, l\}$ *and* $r_2 \in C$*. The atoms undecided in* r_2 *with respect to* S_1 *are* $\{h, j, k\}$*. Each of these atoms occur only once in* r_2*. For each of these atoms, we have* $nc[h, C] = 0$*,* $nc[j, C] = 0$*,* $nc[k, C] = 0$ *and they are increased by one and become* $nc[h, C] = 1$*,* $nc[j, C] = 1$ *and* $nc[k, C] = 1$*. Since,* $nc[h, C'] = 1$*,* $nc[j, C'] = 1$ *and* $nc[k, C'] = 1$*, we get,* $cp[C, C']$ *eventually increased by 3 and* $cp[C, C']$ *becomes 13. The lower bound* $lb[C, C']$ *gets updated to* $8+4-1$ *that is* 11*, where* 4 *is the lower bound of* r_2*. We see that the values of* $cp[C, C']$ *and* $lb[C, C']$ *are backupdated to the values calculated in example 3.6. The conditions* $\upsilon(cp, S_0)$*,* $\upsilon(cp, S_0)$ *and* $\upsilon(cp, S_0)$ *are maintained.*

3.2.5.3 The new_backtrack function

The function **new_backtrack** is illustrated in Figure 3.13. It takes as input a program Π, a stack S, the tables cp, nc, lb, and a boolean variable $found$. The following steps are executed until either S is empty or it finds a picked literal x in S.

function new_backtrack(**var** Π: Prog, **var** S: stack, **var** cp, nc, lb : tables,

var found : bool) : lit

% **precondition :** • $\Pi_0 = r(\Pi_0, S_0)$, $\upsilon(cp, S_0)$, $\upsilon(nc, S_0)$, $\upsilon(lb, S_0)$

% **postcondition :** • If $S_0 = S_1\ x\ S_2$, where x is a picked literal, and the top

% S_2 contains no picked literals, then $S = S_1$ and $\Pi = r(\Pi, S)$, $found$ is true

% and returns $not\ x$. Otherwise $found$ is false.

% • The tables satisfy the conditions $\upsilon(cp, S), \upsilon(nc, S), \upsilon(lb, S)$.

VAR $x : lit$; $R : set\_of\_rules$;

repeat

$x := pop(S)$;

$\Pi := r(\Pi_g, S, R)$; % Π_g is the original program and is global.

$back\_update1(cp, nc, lb, x, S)$;

$back\_update2(cp, nc, lb, R, x, S)$;

until $S = \emptyset$ **or** $x = picked\_literal$

if $S = \emptyset$ **and** $x \neq picked\_literal$ **then** $found := false$;

return *not* x;

end function

Figure 3.13: new_backtrack function

1. A literal x is popped from S.

2. The reduced form of Π with respect to S is found, and any rules which became active because of the removal of x from S are stored in R.

3. *back_update*1 updates the values of the arrays occurring due to change in S.

4. *back_update*2 updates the values of the arrays occurring due to rules which became active. The conditions $\upsilon(cp, S)$, $\upsilon(nc, S)$ and $\upsilon(lb, S)$ are restored.

If *new_backtrack* finds a picked literal in S then it returns the negation of the literal. Otherwise, $found$ is set to false.

Proposition 3.18 *If new_backtrack sets $found$ to false then there is no stable model of Π compatible with B, where B is the input set of literals to new_smodels.*

3.2.6 Proof (sketch)

Proof of correctness of *new_smodels* algorithm: First, we will show that post-conditions of *expand* and *backtrack* of *smodels* are satisfied by *new_expand* and *new_backtrack* of *new_smodels*. The former follows from Propositions (3.7), (3.15), (3.16), and (3.17). The latter follows from (3.18).

Since these are the only routines of *new_smodels* different from the corresponding routines of *smodels*, correctness of *new_smodels* can now be established by the argument used by Niemela and Simons in their proof of correctness of *smodels* [13].

CHAPTER IV

EXPERIMENTAL RESULTS

To experimentally investigate the efficiency of the Extended Evaluation Rule, we compared the performance of *Surya*, with a system called $Surya^{-}$, obtained from *Surya* by removing the implementation of the EE inference rule. The experiments were run on a Sun Ultra 10, with 256 MB of memory.

Currently, there is no established set of problems, or "benchmarks," for testing logic programming systems. Therefore, for our experiments we decided to use typical problems in the logic programming arena, including some which come as part of the distribution of several systems, such as The Queens Problem. Since the Extended Evaluation Rule was designed to be applied to programs containing cardinality, and/or choice rules, the first set of 14 programs utilized in our experiments are written using cardinality and choice rules (normally such programs are faster than those which do not use such rules). The complete description of these problems and their solutions is given in Appendix A. Table 4.1 shows the performance of *Surya* and $Surya^{-}$ on computing the stable models for these programs.

Before discussing the results presented in Table 4.1, we need to recall the intuition behind the "choice points" of the generic *smodels* algorithm, since this is one of the parameters we use in our comparison. As explained in Section 3.1.4, whenever the *pick* function selects, or picks a literal undefined on the stack S of literals during the computation of a stable model, the number of choice points is increased by one. The number of choice points may be used as a rough estimate to help determine the size of the search space for the computation. The greater the number of choice points, the greater the search space involved, and normally the slower the computation. Choice points are not the only deterministic factor for comparing the efficiency of these systems, but they are a significant factor. How to effectively compare these systems is still an open question.

The first column of the table, gives the problem for which the systems are evaluated. The next columns give the number of *choice points*, and total execution *time*, in seconds, for the *Surya* and $Surya^{-}$ systems, respectively. The timing and choice points are given for the first model computed if one exists. Otherwise, we report the timing and choice points needed to establish the absence of the model. The two pigeon programs, and the party program are the only ones with no stable models.

The *Surya* system performed better on the first nine programs in Table 4.1. Our analysis indicates that this behaviour is due to the addition of the Extended Evaluation Rule. We see that on average, for the first nine problems *Surya* is 96% faster than $Surya^{-}$. The number of choice points used by *Surya* is significantly smaller than the number of choice points used by $Surya^{-}$, which implies a decrease in the search space for the computation of a stable model. Another interesting observation in these experiments, is that while there exists a huge discrepancy in the number of choice points utilized by the systems, they compute the same stable model. This means that *Surya* is able to infer more at each choice point, thereby removing subtrees of the search space which do not derive a stable model. Since all the implementations of *Surya* and $Surya^{-}$ are the same, except for the new inference rule, we conclude that the search space decreases as a result of the Extended Evaluation Rule. Therefore, EER is responsible for a substantial increase in the efficiency of the computation of a stable model for these programs.

The remaining five problems on Table 4.1 exemplify the case when the Extended Evaluation Rule does not help to improve the efficiency of the computation. First, we note that *Surya* and $Surya^{-}$ have the same number of choice points for these five problems. Hence the search space is the same for both systems, indicating that additional inferences could not be made by *Surya* with the EER. There are several reasons why this is the case. One reason is that the four (original) inference rules were enough to infer as much as possible in these cases, and no extra information could be derived by considering several rules of the program simultaneously. Another reason is that as of now, the EER has been implemented only for simple cardinality rules and simple choice rules (as defined in sections 3.2.2), and although these programs contain both choice and cardinality rules, these rules are not of this form. Hence the

EER is not applicable in these problems. Next, we see that for these five problems, *Surya* is slightly slower than $Surya^{-}$. This is caused by the overhead due to the extra code for implementing the EER in *Surya*. We believe that this difference can be substantially decreased with a better implementation. On average, for these five problems, *Surya* is 2% slower than $Surya^{-}$. Overall, when comparing the efficiency gains against the losses, we conclude that the addition of the Extended Evaluation Rule is strongly beneficial for the problems analyzed, and we believe that this is the case for a large number of problems which can be represented using cardinality and choice rules.

The next question which needs to be answered is whether or not a significant overhead is caused by the addition of the EER inference rule for programs without cardinality or choice rules. To address this point, we ran another 14 problems on *Surya* and $Surya^{-}$ where the EER could not be applied. The description of these problems and their solutions can be found at the Smodels web site: http://www.tcs.hut.fi/pub/smodels/tests/ by downloading files lp-csp-tests.tar.gz and cp99.tar.gz. These programs consist only of simple rules from $\mathcal{SL}$. Table 4.2 gives the experimental results of *Surya* and $Surya^{-}$ for these programs.

On Table 4.2, one observes that *Surya* and $Surya^{-}$ utilized the exactly same number of choice points for all programs when searching for a stable model. Hence, it is clear that in these examples the EER does not negatively influence this parameter.

The execution times shown in Table 4.2 are for the first model found. On average, *Surya* is 3.6% slower than $Surya^-$ for these programs. This is due to the extra code needed for the implementation of EER. As we mentioned before, we believe that this overhead can be reduced further with a better implementation of the EER. The experiments of Table 4.2 lead us to conclude that although there exists some overhead caused by the introduction of the EER rule, its impact on the efficency of the computation is insignificant and does not invalidate the positive effects that can be achieved with such rule.

We are interested in showing one more point with our experiments. To make sure that the improvement in efficiency we achieved with *Surya*, does not depend on the details of our implementation, we ran the same examples from Table 4.1 on the *Smodels* system. [1]

For the problems given on Tables 4.1 and 4.3, $Surya^-$ is 56% slower than *Smodels*. The reason is that $Surya^-$ does not contain all of the optimizations that *Smodels* implements. For instance, the *Smodels* dependency graphs and source pointers used for the computation of upper closure are not yet implemented in $Surya^-$. Another reason is that the *heuristic* functions used by the Pick functions of *Smodels* and $Surya^-$ are not the same. Note that there is a difference in the number of choice points between $Surya^-$ and *Smodels* for these programs, indicating the Pick

[1] The *Smodels* system is available for downloading at http://www.tcs.hut.fi/Software/smodels.

functions select different literals. We also observe that for small instances of a program *Surya*$^-$ performs as well as *Smodels*. However when the instances grow, there is a large increase in the number of choice points and *Surya*$^-$ becomes significantly slower when compared to *Smodels*. On the other hand, when comparing the *Surya* and *Smodels* results present in Tables 4.1 and 4.3, we find that on average *Surya* is 84% faster than *Smodels* for these problems. These observations demonstrate that the improvement seen in *Surya*, caused by EER, is not implementation specific. Finally, it is clear that by improving the implementation of *Surya*$^-$, it is possible to improve the *Surya* system further.

Table 4.1: Experimental Results - *Surya* & *Surya*$^{-}$ Systems

Problems with card/choice rules	**Surya**		**Surya^{-}**	
	Choice Pts	**Time sec**	**Choice Pts**	**Time sec**
15-queens	72	2.80	5769	39.27
16-queens	532	17.07	89422	637.80
17-queens	236	10.59	71550	658.27
18-queens	1666	68.47	912129	8460.36
9-pigeons	0	0.00	120959	98.59
10-pigeons	0	0.02	1209599	1093.37
11-latin	88	16.46	1891	119.98
12-latin	278	74.40	2877	596.76
5/4-party	0	0.02	-	>24hr
color2.lp & p100	31	2.08	31	1.96
color2.lp & p300	1020	869.41	1020	836.13
wire_route n=10	11	2.97	11	3.03
Knights_knaves	0	0.00	0	0.00
martian_venetians	0	0.00	0	0.00

Table 4.2: Experimental Results - *Surya* & *Surya*$^{-}$ Systems

Problems without card/choice rules	**Surya**		**Surya^{-}**	
	Choice Pts	**Time sec**	**Choice Pts**	**Time sec**
CAR.lp	6	0.01	6	0.01
carx2.lp	4	0.03	4	0.02
mixer.lp	4	0.00	4	0.01
monitor.lp	6	0.04	6	0.03
color.lp & p25	11	0.21	11	0.21
color.lp & p30	11	0.30	11	0.30
pigeon.lp p=7 h=6	715	1.75	715	1.68
pigeon.lp p=8 h=7	6785	20.00	6785	18.94
pigeon.lp p=9 h=8	72091	243.58	72091	233.65
13-queens	7	1.68	7	1.62
15-queens	22	5.63	22	5.48
18-queens	773	219.85	773	212.48
schur.lp n=35 b=5	28	6.15	28	5.94
schur.lp n=40 b=5	34	9.61	34	9.34

Table 4.3: Experimental Results - Smodels System

Problems with card/choice rules	**Smodels**	
	Choice Points	**Time seconds**
15-queens	12947	18.07
16-queens	68831	102.93
17-queens	356277	578.31
18-queens	848300	1506.05
9-pigeons	101941	38.62
10-pigeons	987767	399.90
11-latin	-	>24hr
12-latin	-	>24hr
5/4-party	719681049	approx. 60hr
color2.lp & p100	38	0.54
color2.lp & p300	92	4.58
wire_route n=10	15	0.33
Knights_knaves	0	0.00
martian_venetians	0	0.00

CHAPTER V

CONCLUSIONS

While studying the New Year's Party problem, we realized that the four inference rules of the *Smodels* algorithm were not enough to ensure efficient computation in cases where there existed implicit information distributed among several rules of a program. The lack of an inference rule that would take into account such information resulted in slower computation of models of the program. It became clear that additional inference rule(s) were needed to tackle such cases. Therefore, we planned to add a new inference rule to the smodels algorithm called the Extended Evaluation Rule. To achieve this objective, we developed:

- a new algorithm, called *new_smodels* algorithm, incorporating EER.

  The algorithm does not merge rules or extend the program with these rules as mentioned in the definition of EER. Instead, the algorithm efficiently uses three bookkeeping tables (as described in section 3.2.3.1), and keeps track of the information needed from rules of the program in these tables. As explained before, only the number of complementary pairs and the lower bound of each merged rule is maintained and EER is applied by comparing them. This greatly increases the efficiency of the algorithm with very low overhead.

- System *Surya* was implemented based on the *new_smodels* algorithm.

The implementation is written in C. The program uses the $lparse$ frontend of $Smodels$ system. It also uses another grounder which collects the constraint sets of the program Π and also finds the related constraint sets.

- The efficiency of the system was evaluated by comparing it with a system called $Surya^{-}$ which was obtained by removing the implementation of EER in $Surya$.

  We found that for programs where EER helped, the increase in efficiency of $Surya$ was considerable (96%). For other programs, $Surya$ had a low overhead when compared to $Surya^{-}$ (3%). We also found that $Surya^{-}$ was 56% slower than $Smodels$ for the problems tackled in experiments.

- The sketch of the proof of correctness for the algorithm is presented in section 3.2.6.

Extra space needed for the implementation of EER is only to store the three bookkeeping tables and $\mathcal{CS}$, which consists of sets of sets of cardinality rules of the program. For $\mathcal{CS}$, it suffices to maintain only the information regarding which rule belongs to which constraint set in $\mathcal{CS}$ and not the rule itself. The bookkeeping tables *cp* & *lb* uses a space of size $N \times N$ where N is the number of constraint sets in $\mathcal{CS}$ of the program. The number of constraint sets in $\mathcal{CS}$ of the program is approximately equal to twice the number of simple cardinality rules 3.8 and simple choice rules 3.4 of the program input to $lparse$. This number is very small, even for huge programs, and depends on the problem and not its instance. The table *nc* uses $A \times N$ space, where A is the number of atoms in $\mathcal{CS}$ and N is the number of constraint sets in $\mathcal{CS}$. A depends on the instance of a program.

Based on the experiments ran, we conclude that the *Extended Evaluation Rule* in some cases is very helpful in decreasing the search space, thereby increasing the efficiency in the computation of stable models for programs containing simple choice and/or cardinality rules. On the other hand, it does not cause much overhead for programs not containing simple choice or cardinality rules.

Smodels is currently the most efficient implementation for computing stable models of logic programs. Although *Surya*'s implementation is still not comparable to *Smodels*' for a general class of programs, we believe that the efficiency of *Smodels* would considerably increase for programs containing choice and/or cardinality rules, if the *EER* inference rule would be incorporated to it.

5.1 Future Work

Presently, *Surya* does not have all the optimization techniques implemented in the *Smodels* system and it also does not allow weight rules. Immediate improvements I plan to work on for the *Surya* system include the implementation of the optimizations used by *Smodels* and the implementation of the weight rules available for the Smodels Language. The Extended Evaluation Rule can be naturally generalized for use with weight rules. Therefore, the implementation of the EER inference rule on weight rules is a natural step to be taken.

The work on this thesis shows that there exists extra information implicitly distributed among rules of a logic program which is not always explicitly stated by single rules. This realization led to the design and implementation of a new algorithm and system for computation of stable models of logic programs using a original new inference rule called Extended Evaluation Rule. We are interested in pursuing this work further and investigate other programs to check for other types of hidden relationships among rules that could lead to new developments and increase of efficiency of the computation of such programs.

REFERENCES

[1] T. Dell'Armi, W. Faber, G. Ielpa, C. Koch, N. Leone, S. Perri, and G. Pfeifer. System Description: DLV. In *Logic Programming and Nonmonotonic Reasoning — 6th International Conference, LPNMR'01*, in AI (LNAI), 2173:409-412, 2001.

[2] M. Balduccini, M. Barry, M. Gelfond, M. Nogueira, and R. Watson. An A-Prolog decision support system for the Space Shuttle. *Lecture Notes in Computer Science-Proceedings of Practical Aspects of Declarative Languages'01*, (2001), 1990:169-183.

[3] C. Baral and M. Gelfond. Reasoning Agents in Dynamic Domains. in *Logic Based Artificial Intelligence.* Kluwer, Boston, 2000.

[4] Y. Dimopoulos, N. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. *Lecture Notes in Artificial Intelligence - Recent Advances in AI Planning, Proc. of the 4th European Conference on Planning, ECP'97*, 1348:169-181, 1997

[5] D. East and M. Truszczynski. More on Wire Routing with ASP. *Technical Report in Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, 2001 AAAI Spring Symposium* 39-44, 2001.

[6] E. Erdem, V. Lifschitz, and M. Wong. Wire routing and satisfiability planning. *Proc. of CL-2000*, 822-836,2000.

[7] M. Gelfond. Representing Knowledge in A-Prolog. To appear in Computational Logic: from logic programming to the future, collection of papers in honour of B. Kowalski.

[8] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.,* pages 1070-1080, 1988.

[9] R. Kowalski. *Logic for Problem Solving.* North-Holland, Amsterdam, 1979.

[10] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, 375-398, Spring-Verlag, NewYork, 1999.

[11] J. McCarthy. Elaboration Tolerance. Manuscript, 1997.

[12] I. Niemela and Patrik Simons. Extending the Smodels System with Cardinality and Weight Constraints. In *Logic-Based Artificial Intelligence 2000*, 491-521.

[13] P. Simons. Extending the stable model semantics with more expressive rules. In *5th International Conference, LPNMR'99*, 305-316.

[14] P. Simons. Extending and implementing the stable model semantics. Research Report A58, Helsinki University of Technology, April, 2000.

[15] T. Soininen and I. Niemela. Developing a declarative rule language for applications in product configuration. *In Practical Aspects of Declarative Languages*, LNCS 1551, pages 305-319, 1999.

[16] T. Syrjanen. Implementation of local grounding for logic programs with stable model semantics. Technical Report B18, Helsinki University of Technology, October, 1998.

[17] T. Syrjanen. Lparse User's Manual available at http://www.tcs.hut.fi/Software /smodels/

[18] System built by M. Balduccini at Texas Tech University.

[19] http://www.cs.utexas.edu/users/vl/tag/

[20] http://www.cs.utexas.edu/users/tag/

[21] http://www.krlab.cs.ttu.edu/

[22] Problem and Discussions on Seating Arrangements available at http://www.cs.utexas.edu/users/vl/tag/discussions.html

[23] http://www.tcs.hut.fi/Software/smodels/

APPENDIX

We give all the programs used in Tables 4.1 and 4.3 in this appendix.

N-queens problem

N-Queens is a famous problem, where given N queens, the problem consists in finding a placing for each queen in a $N \times N$ chess board such that no two queens can attack each other. A natural solution for such a problem in Smodels language is written as follows:

% Number of columns/rows in a chess board is equal to n.

$c(1..n).$

% Choose n columns for the n queens (the rows are numbered same as queens).

$n \ \{ \ at(Q, C) : c(Q) : c(C) \ \} \ n.$

% No two queens can be in the same row.

$:- \ 2 \ \{ \ at(Q, C) : c(C) \ \}, \ c(Q).$

% No two queens can be in the same column.

$:- \ 2 \ \{ \ at(Q, C) : c(Q) \ \}, \ c(C).$

% No two queens can be in the same diagonal.

$:- \ at(Q1, C1), \ at(Q2, C2), \ c(Q1), \ c(C1), \ c(Q2), \ c(C2),$

$\quad neq(Q1, Q2), \ abs(Q1 - Q2) == abs(C1 - C2).$

N-pigeons problem

N-pigeons is also a famous problem where $n+1$ pigeons need to be alloted to n holes such that no two pigeons can be placed in the same hole. It is easy to see that such a problem does not have any solutions. The following program was used to run on *Surya*, *Surya*$^-$ and *Smodels* systems. The program was written by Ilkka Niemela and it is publicly available at www.tcs.hut.fi/ ini/esslli99/ lecture5.ps

% Number of pigeons is equal to n+1.

$pigeon(1..n+1)$.

% Number of holes is equal to N.

$hole(1..n)$.

% Place one pigeon in one hole.

$1\ \{\ in(P,H) : hole(H)\ \}\ 1 :- \ pigeon(P)$.

% No two pigeons can have the same hole.

$:-\ 2\ \{\ in(P,H) : pigeon(P)\ \},\ hole(H)$.

Latin Squares problem

Given n numbers, the problem consists in filling a $n \times n$ matrix with the numbers, such that a number should not appear again on the same row or column. The following program is written in Smodels language.

% There are n numbers.

$l(1..n).$

% For each number X find n positions in the matrix

$n\ \{\ at(X,R,C) : l(R) : l(C)\ \}\ n :- l(X).$

% Two numbers cannot occupy the same row and column.

$:- 2\ \{\ at(X,R,C) : l(X)\ \},\ l(R),\ l(C).$

% A number cannot be in the same row twice.

$:- 2\ \{\ at(X,R,C) : l(R)\ \},\ l(X),\ l(C).$

% A number cannot be in the same column twice.

$:- 2\ \{\ at(X,R,C) : l(C)\ \},\ l(X),\ l(R).$

New Year's Party problem

New Year's Party problem was posed by Dr. Vladimir Lifschitz to the Texas Action Group [19] members to solve. The problem is as follows:

You are organizing a New Year's Eve party. There will be N tables in the room, with M chairs around each table. You need to select a table for each of the guests, so that two conditions are satisfied : (1) Some guests like each other and are to be seated in the same table. (2) Some guests dislike each other and are to be seated in different tables. The number of guests are $M \times N$.

Here is a program representing the problem taken from TAG technical discussions [22].

% inputs used in Tables 4.1 and 4.3

const chairs = 4.

const tables = 5.

const guests = chairs * tables.

likes(1,2). dislikes(2,1).

% The number of tables and guests.

table(1..tables).

guest(1..guests).

% If X likes Y then Y likes X. Similarly for dislikes.

$likes0(X, Y) :- likes(X, Y).$

$likes0(X, Y) :- likes(Y, X).$

$dislikes0(X, Y) :- dislikes(X, Y).$

$dislikes0(X, Y) :- dislikes(Y, X).$

% guests who like each other must be at the same table

$:- at(G1, T),\ not\ at(G2, T),\ likes0(G1, G2),\ table(T).$

% guests who dislike each other must not be at the same table

$:- at(G1, T),\ at(G2, T),\ dislikes0(G1, G2),\ table(T).$

% each table must have exactly as many guests as chairs

$chairs\ \{\ at(G, T) : guest(G)\ \}\ chairs :- table(T).$

% no guest can be at more than one table

$:-\ 2\ \{\ at(G,T) : table(T)\ \}\ ,\ guest(G).$

Coloring problem

The following coloring problem was taken from *lparse* manual [23]. Given a graph as a set of vertices and arcs find a way to color the vertices with n colors such that two adjacent vertex are not colored with the same color.

% number of colors

const n=4.

% There are n colors.

$color(1..n).$

% Each vertex should have exactly one color:

$1\ \{\ v\_color(N,C) : color(C)\ \}\ 1 :-\ vertex(N).$

% Two adjacent vertices need to have different colors:

$:-\ v\_color(X,C),\ v\_color(Y,C),\ arc(X,Y),\ color(C).$

The input graph for this problem is $p100$. It is available at http://tcs.hut.fi/pub/smodels/tests/lp-csp-tests.tar.gz.

Wire Routing problem

The wire routing program involves in connecting wires to terminal points on a chip.

The wires thus routed should not overlap with each other and with regions occupied by other components placed on the chip. The following is the program found in [5].

% input corresponds to figure 4 in [5]

const n=10.

pt(1..n).

% blocked regions where other components are placed:

block(4,6). block(4,7). block(4,8). block(4,9).

block(5,6). block(5,7). block(5,8). block(5,9).

block(6,6). block(6,7). block(6,8). block(6,9).

block(8,7). block(8,8). block(8,9). block(9,7).

block(9,8). block(9,9). block(7,3). block(7,4).

block(8,3). block(8,4). block(9,3). block(9,4).

% Three wires need to be routed:

wire(w1). wire(w2). wire(w3).

% Terminal points which the wires need to connect:

terminal(3,2,w1). terminal(9,5,w1). terminal(8,6,w2).

terminal(2,5,w2). terminal(7,8,w3). terminal(2,8,w3).

% More than one wire cannot pass through a point.

$:- 2 \ \{ \ path(I, J, W) : wire(W) \ \}, \ pt(I; J).$

% To prevent more than two adjacent points of any point in a path from being included.

$1 \ \{ \ path(M,N,W) : pt(M) : pt(N) : eq((abs(I-M)+abs(J-N)),1) \ \} \ 1 :-$

$endpoint(I,J,W), \ wire(W), \ pt(I;J).$

% Exactly one adjacent point for each terminal point is included.

$2 \ \{ \ path(M,N,W) : pt(M) : pt(N) : eq((abs(I-M)+abs(J-N)),1) \ \} \ 2 :-$

$path(I,J,W), \ not \ endpoint(I,J,W), \ wire(W), \ pt(I;J).$

% Wires cannot go over blocked regions.

$:- path(I,J,W), \ block(I,J), \ pt(I;J), \ wire(W).$

% Terminal points are to be included in the path.

$endpoint(I,J,W) :- terminal(I,J,W).$

$path(I,J,W) :- terminal(I,J,W).$

% Prohibiting one block cycle.

$:- path(I,J,W), \ path(I+1,J,W), \ path(I,J+1,W),$

$path(I+1,J+1,W), \ pt(I), \ pt(J), \ wire(W).$

Knights & Knaves

The description of the problem is as follows: The island of Knights and Knaves has two types of inhabitants: knights, who always tell the truth, and knaves, who always lie.

One day, three inhabitants (A, B and C) of the island met a foreign tourist and gave the following information about themselves: (1) A said that B and C are both knights. (2) B said that A is a knave and C is a knight. what types are A, B and C?

Here is the representation of the problem taken from [17].

% Each person is either a knight or a knave.

$1\ \{\ knight(P),\ knave(P)\ \}\ 1 :-\ person(P).$

% There are three persons in the puzzle:

$person(a; b; c).$ % Rest of this program models the two hints.

% Hint 1:

% If A tells the truth, B and C are both Knights.

$2\ \{\ knight(b),\ knight(c)\ \}\ 2 :-\ knight(a).$

% If A lies, both cannot be knights.

$:-\ knave(a),\ knight(b),\ knight(c).$

% Hint 2:

% If B tells the truth, A is a knave and B is a knight.

$2\ \{\ knave(a),\ knight(c)\ \}\ 2 :-\ knight(b).$

% If B lies, one of the claims has to be false.

$:-\ knave(b),\ knave(a),\ knight(c).$

Martian - Venusian Club

On Ganymede - a satellite of jupiter - there is a club known as the Martian - Venusian Club. All members are either from Mars or from Venus, although visitors are sometimes allowed. An earthling is unable to distinguish Martians from Venetians by their appearance. Also earthlings cannot distinguish either Martian or Venusian males from females, since they dress alike. Logicians, however, have an advantage, since the Venusian women always tell the truth and the Venusian men always lie. The martians are the opposite; the Martian men tell the truth and the Martian women always lie. One day a visitor met two club members, Ork and Bog, who made the following statements:

1. Ork: Bog is from Venus.

2. Bog: Ork is from Mars.

3. Ork: Bog is male.

4. Bog: Ork is female.

where are Ork and Bog from, and are they male or female?

The following is the program representing the problem from [17].

% All persons are from Mars or Venus

$1 \; \{ \; martian(P), \; venetian(P) \; \} \; 1 :- person(P).$

% All persons are male or female

$1 \; \{ \; female(P), \; male(P) \; \} \; 1 :- person(P).$

% all persons either lie or tell teh truth depending

% on their origin and sex.

$lies(P) :- person(P),\ martian(P),\ female(P).$

$lies(P) :- person(P),\ venetian(P),\ male(P).$

$truthful(P) :- person(P),\ martian(P),\ male(P).$

$truthful(P) :- person(P),\ venetian(P),\ female(P).$

% a person may not tell the truth and lie at the same time.

$:- person(P),\ lies(P),\ truthful(P).$

% persons:

$person(ork; bog).$

% Hints

% 1.

$venetian(bog) :- truthful(ork).$

$:- lies(ork),\ venetian(bog).$

% 2.

$martian(ork) :- truthful(bog).$

$:- lies(bog),\ martian(ork).$

% 3.

$male(bog) :- truthful(ork).$

$:- lies(ork),\ male(bog).$

% 4.

$$female(ork) :- truthful(bog).$$

$$:- lies(bog),\ female(ork).$$