University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2003

Computing stable models of logic programs

Soumya Singhi
University of Kentucky, soumya-singhi@yahoo.com

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Singhi, Soumya, "Computing stable models of logic programs" (2003). *University of Kentucky Master's Theses*. 223.
https://uknowledge.uky.edu/gradschool_theses/223

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

Computing stable models of logic programs

Solution of any search problem lies in its search space. A search is a systematic examination of candidate solutions of a search problem. In this thesis, we present a search heuristic that we call *cr-smodels*. *cr-smodels* prunes the search space to quickly reach to the solution of a problem.

The idea is to pick an atom for branching, that lowers the growth rate of the linear recurrence and thus, minimizes the remaining search space. Our goal in developing *cr-smodels* is to develop a search heuristic that is efficient on a wide range of problems.

Then, we test *cr-smodels* over wide range of randomly generated benchmarks. We observed that often randomly generated graphs with no Hamiltonian cycle were trivial to solve. Since, Hamiltonian cycle is an important benchamrk problem, my other goal is to develop techniques that generates hard instances of graphs with no Hamiltonian cycle.

KEYWORDS: Stable model Semantics, *smodels*, Search Algorithm,
Declarative programming, Logic programming

Soumya Singhi

Date

Computing stable models of logic programs

By

Soumya Singhi

Dr. Mirosław Truszczyński
____
(Director Of Thesis)

Dr. Grzegorz W. Wasilkowaski
____
(Director Of Graduate Studies)

Date
____

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part requires also the consent of the Dean of The Graduate School of the University of Kentucky.

THESIS

Soumya Singhi

The Graduate School
University of Kentucky
2003

Computing stable models of logic programs

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Masters of Science
in the College of Engineering
at the University of Kentucky

By

Soumya Singhi

Lexington, Kentucky

Director: Dr. Mirosław Truszczyński, Professor of Computer Science

Lexington, Kentucky

2003

Copyright ©Soumya Singhi 2003.

Acknowledgments

First, I want to thank my Thesis Chair, Dr. Mirosław Truszczyński, for his continuous guidance and support. This includes not only the period of the creation of this Thesis, but also throughout my master's studies and research at the University of Kentucky. His faith in me and the freedom he provided have made this work possible.

I would also like to thank Dr. Raphael Finkel for providing his programs that were useful in my experimentation. Assistance from my parents and other family members was extremely helpful during my stay here. I also thank to all my professors at the University of Kentucky for providing me with an enriching experience. I would also like to thank staff members of Computer Science department and all my colleagues for their direct or indirect support in completing my Thesis.

Table of Contents

Acknowledgements **iii**

List of Figures **vi**

List of Files **viii**

1 Introduction **1**
1.1 Declarative Programming 2
1.2 Horn Logic Programming 2
1.3 Answer Set Programming 5
1.4 ASP vs standard logic programming 8
1.5 Contribution 9
1.6 Outline of Thesis 9

2 Stable Logic Programming **11**
2.1 Stable Model Semantics for Propositional Logic Programs 11
2.2 Lifting semantics of SLP for programs with variables 13
2.3 Extension of semantics 15
2.4 Computing Stable Models 16

3 Smodels **18**
3.1 Overview of the algorithm 18
3.2 lookahead 20
3.3 Heuristic 22
3.4 Problem Statement 23

4 New Heuristic **25**
4.1 Estimation of search space 25
4.2 Computing roots of the equation 26

5 Generating hard unsat HC instances **28**
5.1 Expanding with 8-spoiler graph 29
5.2 Ring Join 31
5.3 Graphs used for n-ring join 33

6 Experimentation **40**
6.1 Smodels 40
6.2 Generating random graphs 41
6.3 Results for selected problems 41
6.3.1 Hamiltonian cycle 42
6.3.2 Graph 3-Coloring 44
6.3.3 N-Queen 45
6.3.4 n-queen.x 45
6.3.5 Vertex Cover 46
6.4 Summary of Results 47

7 Discussion **48**

A Logic Programs **49**
A.1 Hamiltonian Cycle problem 49
A.2 Graph 3-color 49
A.3 N-queen 50
A.4 N-queen.x 50
A.5 Vertex Cover 51

Bibliography **52**

Vita **55**

List of Figures

1.1 Proof to dertermine membership of an atom in least Herbrand Model . . . 6
1.2 Graph with 3 vertices and 2 edges [ASP6] 7

3.1 Algorithm for lookahead 21
3.2 Reduction of search space with lookahead 22

4.1 Estimation of search space. 26
4.2 Newton-Raphson Method. 27

5.1 8-Spoiler graph 30
5.2 Spoiler expansion of graph G 30
5.3 n-ring join of a graph G 32
5.4 42-vertex nonhamiltonian graph 33
5.5 42-vertex nonhamiltonian graph 34
5.6 42-vertex nonhamiltonian graph 34
5.7 44-vertex nonhamiltonian graph 34
5.8 46-vertex nonhamiltonian graph 35
5.9 50-vertex nonhamiltonian graph 35
5.10 50-vertex nonhamiltonian graph 36
5.11 50-vertex nonhamiltonian graph 36
5.12 52-vertex nonhamiltonian graph 36
5.13 52-vertex nonhamiltonian graph 37
5.14 52-vertex nonhamiltonian graph 37
5.15 52-vertex nonhamiltonian graph 37
5.16 52-vertex nonhamiltonian graph 38
5.17 52-vertex nonhamiltonian graph 38
5.18 44-vertex nonhamiltonian graph 38
5.19 46-vertex nonhamiltonian graph 39
5.20 46-vertex nonhamiltonian graph 39
5.21 48-vertex nonhamiltonian graph 39

6.1 A algorithm to generate random graphs 41
6.2 Comparison of *smodels* and *cr-smodels* on Hamiltonian cycle problem . . . 42

6.3 Comparison with non-Hamiltonian graphs generated using "Expanding with 8-spoiler" technique. 43
6.4 Comparison with non-Hamiltonian graphs generated using "n-ring join" technique. 44
6.5 Comparison of *smodels* and *cr-smodels* on graph 3-coloring problem 45
6.6 Comparison of *smodels* and *cr-smodels* on N-queen problem 46
6.7 Comparison of *smodels* and *cr-smodels* on n-Queen.x problem 47

List of Files

1. SSThesis.pdf 795KB

Chapter 1

Introduction

Computer programming languages can be divided into two categories, imperative languages and declarative languages.

Imperative programming is based on a statement-at-a-time paradigm where each statement has some effect on a memory store. Kowalski [1] argues that programs written in imperative languages are generally large, hard to write, debug, and maintain because imperative programming lays much stress on "how" a solution procedure is specified. Examples of imperative languages are Pascal, C and Java, etc.

In contrast, declarative programming is based on stating the relationship between inputs and outputs. A program in a declarative programming language describe the constraints of the problem [17]. The solutions to the problem are computed with a separate "solver program". Examples of declarative languages are ML, pure Lisp and pure Prolog.

In short, a program in a declarative language provides specifications of the problem to solve, while a program in imperative language specifies "how" to solve a problem [24]. Declarative programs are often short, easy to write, debug, and maintain. However, execution speed of declarative programs are often slower than that of imperative programs.

Logic programming with stable model semantics has evolved as the novel paradigm in declarative programming [24]. *smodels* [13] is among the best available implementation of this paradigm. In this work, we will study the algorithms for the solver program *smodels* that computes stable models of finite propositional logic programs.

1.1 Declarative Programming

A program in a declarative programming language is a *theory* in some logic and is made up of formulas (usually clauses). A formula in a program specifies a constraint of the problem to be solved. Every program written in declarative language has a precise meaning provided by the semantics of the underlying logic that specifies *interpretations* (models) of the program.

Declarative programming provides programmers with a higher-level abstraction in writing programs. The programmers concentrate on stating "what" is to be computed, not necessarily "how" it is to be computed. Consequently, declarative programming carries with itself a promise of easier code development, facilitates modular design of software and eases the problem of program verification [24]. It is due to separation of logic from the control of the program.

The semantics of a logic program provides mapping between interpretations (models) of the logic program and solutions of the problem to solve. The two paradigms of declarative programming based on these mapping techniques that are also relevant to my work are as follows: Horn logic programming, with its formal basis in first-order logic, (examples are *Prolog, Eclipse*) and Answer set programming (examples are *smodels, dlv, aspps*).

1.2 Horn Logic Programming

Horn logic programming is an important formalism of declarative programming paradigm and is also in some respects relevant to my work. Therefore, we will briefly describe Horn logic programming. A more detailed presentation may be found in many logic texts, for instance is [6]. We start with defining the *language* of Horn logic. A language in Horn logic consists of:

1. Countable sets of constant, variable, function and predicate symbols,
2. Propositional connectives (like conjunction and disjunction) and constants,
3. The quantifier symbol $\forall$,
4. Punctuation symbols: parenthesis and comma.

Each function and predicate symbol in the language has an integer arity. The *terms* in the language are constructed recursively from its constant, variable, function symbols and punctuation symbols. For example *f(X,Y)* is a term built of 2-ary function symbol f and two variables X and Y. $f(f(X,Y),Y)$ is another example of a term. The *atoms* in the language are expressions of the form, $p(t_1,...,t_k)$, where $t_1,...,t_k$ are terms in the language and p is a predicate. Each *Horn clause* is a universally quantified formula of form

$$\forall y_0,...,y_m \; p(t_0) \leftarrow q_1(t_1) \wedge ... \wedge q_k(t_k) \tag{1.1}$$

where p and $q_i s$ are predicate of the language, t_i, $0 \leq i \leq k$ are the tuples of terms of appropriate arity and y_i, $0 \leq i \leq m$, are all variables that appear in these terms. In our notation, we follow the standard of logic programming practice by dropping the universal quantifier. Thus (1.1) can be written as

$$p(t_0) \leftarrow q_1(t_1) \wedge ... \wedge q_k(t_k).$$

A *Horn logic theory* or *program* is a set of Horn clauses.

The *Herbrand universe* $HU(P)$ of a logic program P is the set of all ground terms that can be constructed using function symbols and constants in P. If P does not contain any constants then an arbitrary constant is chosen and $HU(P)$ is based on that constant. If P is function-free then the $HU(P)$ is finite; otherwise, it is infinite. The *Herbrand base* $HB(P)$ of P is the set of all ground atoms that can be constructed using predicates in P and terms in $HU(P)$.

An *instance* of an expression is constructed by uniformly replacing all its variables by the ground terms. The *Herbrand instantiation* of a program P (also referred as $ground(P)$), is a set of all ground instances of rules in P, which may be constructed using terms in $HU(P)$.

An *interpretation* $I(P)$ of a program P is a subset of $HB(P)$. An interpretation assigns a truth value to each element of $HB(P)$. If an atom a is in $I(P)$, then its value is **true** in I, otherwise it is **false**. If $I(P)$ makes every rule in $ground(P)$ **true** then $I(P)$ is a *Herbrand model* of the program. A Herbrand model M_0 is *least* if it is contained in every other model.

Theorem 1.1 *Every Horn program has a unique least Herbrand model.*

Example 1.1 *Consider the following program* π*:*

p(1).

q(2).

q(x) $\leftarrow$ *p(x).*

Herbrand universe, $HU(\pi)$*, is* {*1, 2*}.

Herbrand base, $HB(\pi)$*, is* { *p(1), p(2), q(1), q(2)*}.

Herbrand instantiation, $HI(\pi)$*, is*

p(1).

q(2).

q(1) $\leftarrow$ *p(1).*

q(2) $\leftarrow$ *p(2).*

π *has two Herbrand models:*

{*p(1), q(1), q(2)*} *(1.2)*

and

{*p(1), p(2), q(1), q(2)*}.

The least Herbrand model $(LHM(\pi))$*of the program* π *is (1.2).*

The semantics of a Horn logic depends on the notion of a single intended model. A Horn program P may have many Herbrand models but from Theorem 1.1, one of them is the *least* model (denoted by $LM(P)$) also distinguished as an intended model. We use this model to define the meaning of Horn program. We say that P *expresses* a subset X of Herbrand universe $HU(P)$ if for some predicate p occurring in P,

$$X = \{t \in HU(P) : p(t) \text{ is true in } LM(P)\}.$$

Intuitively, we can say that in Horn programming all the terms may be viewed as a solution to a problem [6]. Terms are complex data structures, thereby, allowing us to recursively define objects. The ability to model recursive definitions is responsible for the expressive power of Horn logic programming. Fundamental to logic programming is the result that any recursively enumerable set can be specified by a Horn program. Thus, Horn programs are as expressive as Turing machines [24].

Theorem 1.2 *[3], [20], A decision problem can be solved by a Horn logic program if and only if it is recursively enumerable.*

In practice, generating *ground(P)* is often cumbersome because even in the case of function-free languages the size of *ground(P)* is exponential in size of P (in all other cases

ground(P) is infinite). Moreover, it is not always necessary to compute the least model of *ground*(P) to determine whether P entails A for some particular atom A. Several proof-based approach based on variants of the *Resolution Principle* of Robinson [1965] can be used instead. SLD-resolution [19] is one such variants.

SLD-resolution is described as follows. A *goal* is a conjunction of atoms, and a *substitution* is a function v that maps variables $v_1, ..., v_n$ to terms $t_1, ..., t_n$. The result of simultaneous replacement of variables v_i by terms t_i in an expression E is denoted by Ev. For a given goal G and a program P, SLD-resolution tries to find a substitution v such that Gv logically follows from P. The initial goal is repeatedly transformed until the empty goal is obtained. Each transformation step is based on the application of the resolution rule to a selected atom B_i from the goal $B_1, ..., B_m$ and a clause $A_0 \leftarrow A_1, ..., A_n$ from P. SLD-resolution tries to *unify* B_i with the head A_0 i.e. to find a substitution v such that $A_0 v = B_i v$. Such a substitution v is called as *unifier* of A_0 and B_i. If a unifier v exists, a most general such v (which is essentially unique) also referred as *most general unifier* (mgu) is chosen and the goal is transformed into

$$(B_1,\ ...,\ B_{i-1},\ A_1,\ ...,\ A_n,\ B_{i+1},\ ...,\ B_m)v$$

Example 1.2 *Let us consider the following set of clauses:*

larger(hammer, feather).

denser(hammer, feather).

heavier(A,B) $\leftarrow$ *larger(A,B), denser(hammer, feather).*

Now we ask a query: Is hammer heavier than feather? (?-heavier(hammer,feather))

Figure 1.1 depicts the SLD-resolution refutation method. It identifies the substitution v = {A / hammer, B/feather }.

1.3 Answer Set Programming

Answer set programming is an alternate way to use logic in declarative programming. To represent a problem, a finite theory is designed so that each *model* of the theory encodes the solution to a problem to solve. Then, the models of the theory are computed to solve the problem. This model-based approach is referred to as *answer set programming* or ASP.

An ASP system allows variables but not function symbols in the language. A *theory* in ASP is a pair (D, P), where D is a set of ground atoms representing an *instance* of the

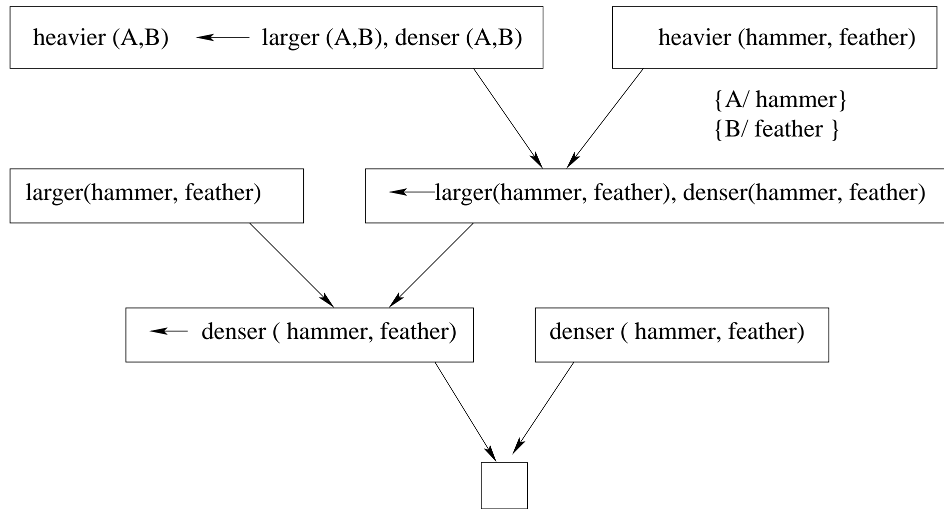

Figure 1.1: Proof to dertermine membership of an atom in least Herbrand Model

problem, and P is a set of clauses representing a program (an abstraction of a problem). The *meaning* of a theory $T = (D, P)$ is given by a *family* of models.

To solve a problem π in ASP formalism, we find a program P so that solutions to π can be reconstructed in polynomial time (ideally linear) from the answer sets to P.

Propositional logic can be viewed as an answer set system because there is a concept of a theory and a theory can be assigned models [6]. We will show how propositional logic can be used as an ASP system to encode the graph coloring problem.

A graph coloring problem is an assignment of colors such that each vertices is assigned exactly one color and the vertices joined with an edge are not assigned the same color. We will now present a propositional theory encoding of a graph coloring problem for the graph shown in Figure 1.3. We will assume three colors are available.

Example 1.3 *Let us consider the graph shown in Figure 1.3. A propositional theory T is constructed by encoding constraints of graph 3-colorability problem. We assume that the only colors available are red, green and blue. We will use the connectives '$\rightarrow$' (implies), '$\vee$' (disjunction), '$\wedge$' (conjunction).*

For the theory T, we assme that data D is some random graph and P as the logic program. We will represent clauses as the implication of the form $a_1 \wedge ... \wedge a_k \rightarrow b_1 \vee ... \vee b_m$, where each a_i and b_k are terms. Below, we will represent a clause as a constraint by the

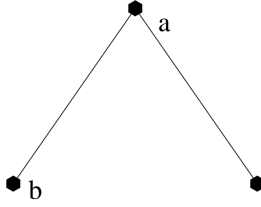

Figure 1.2: Graph with 3 vertices and 2 edges [ASP6]

means of an implication. In a clause, terms to the left of a implication are connected with conjunction, while the terms to the right are connected with disjunction. For a clause to be true, at least one of the terms in conjunction should be false, otherwise, at least one of the terms in disjunction should be true. These clauses are easy to write and understand.

(C1) *color(a,red)* $\vee$ *color(a,green)* $\vee$ *color(a,blue).*

(C2) *color(b,red)* $\vee$ *color(b,green)* $\vee$ *color(b,blue).*

(C3) *color(c,red)* $\vee$ *color(c,green)* $\vee$ *color(c,blue).*

Clauses (C1) -(C3) ensure that each vertex is assigned at least one color

(C4) *color(a,red)* $\wedge$ *color(a,green)* $\rightarrow$.

(C5) *color(a,red)* $\wedge$ *color(a,blue)* $\rightarrow$.

(C6) *color(a,blue)* $\wedge$ *color(a,green)* $\rightarrow$.

(C7) *color(b,red)* $\wedge$ *color(b,green)* $\rightarrow$.

(C8) *color(b,red)* $\wedge$ *color(b,blue)* $\rightarrow$.

(C9) *color(b,blue)* $\wedge$ *color(b,green)* $\rightarrow$.

(C10) *color(c,red)* $\wedge$ *color(c,green)* $\rightarrow$.

(C11) *color(c,red)* $\wedge$ *color(c,blue)* $\rightarrow$.

(C12) *color(c,blue)* $\wedge$ *color(c,green)* $\rightarrow$.

Clause (C4)-(C12) ensure that no vertex is assigned more than one color.

(C13) *color(a,red)* $\wedge$ *color(b,red)* $\rightarrow$.

(C13) *color(a,red)* $\wedge$ *color(c,red)* $\rightarrow$.

(C13) *color(a,green)* $\wedge$ *color(b,green)* $\rightarrow$.

(C13) *color(a,green)* $\wedge$ *color(c,green)* $\rightarrow$.

(C13) *color(a,blue)* $\wedge$ *color(b,blue)* $\rightarrow$.

(C13) *color(a,blue)* $\wedge$ *color(c,blue)* $\rightarrow$.

Clauses (C13)-(C18) specifies the constraints of coloring

Propositional theory T of the graph depicted in Figure 1.3 has twelve answer sets each determining the valid 3-color coloring of G. Here is one such valid coloring

$$\{color(a,red),\ color(b,green), color(c,blue)\ \}$$

Such theories can be proceeded with the SAT solvers [7]. On asking the query "Can the graph G (Figure 1.3) be 3-colored?", a SAT solver will answer *true*.

1.4 ASP vs standard logic programming

The primary feature that answer set programming shares with other logic programming systems is the separation of logic from the control of the program. Despite the fact that answer set semantics has its roots in standard logic programming it differs significantly from standard logic programming systems as follows [24]:

1. An ASP program is a representation of an intended models rather than single intended model as in logic programming.

2. Unlike in Horn logic programming and its extension in several systems, ASP does not provide support for function symbols.

3. In ASP, a term is a variable or a constant, unlike in Horn logic programming, where a term is a complex data structure made up of function symbols, variables and constants.

4. ASP programs provide constraints of problem to solve, while a clause in logic programming specifies recursive definition of an object.

5. The control mechanism in ASP is a backtracking search algorithm that computes models of a program. In standard logic programming, the proof-based techniques are used to determine value of variables (SLD-resolution) to reach the goal or solution to the problem to solve.

6. Typically, ASP has lower expressive power than standard logic programming systems. Still, ASP can solve a wide range of problems and includes all decision problems from class NP and many planning and constraint satisfaction problems that are of importance in artificial intelligence and operations research.

1.5 Contribution

Stable logic programming is an important ASP formalism. *smodels* system is one of the best known implementations of stable model semantics for logic programs. System *smodels* implements a search to compute stable models of a logic program.

The key step in the search, is to select an atom for branching and *smodels* uses a certain heuristic to do so. In this work, we present a new heuristic to select an atom for branching. We call this modification to *smodels* as *cr-smodels*. *cr-smodels* is based on the intuition that the size the remaining search space can be represented as a term in some linear recurrence relation and the estimation of the search space is done by solving this relation. *cr-smodels* picks an atom for branching that minimizes the estimate of the size of remaining search space. The expectation is that branching on such an atom will quickly lead to a solution of the problem.

We compare the performance of *cr-smodels* with *smodels* over a broad range of benchmarks. These benchmarks were generated randomly. The benchmark instances were chosen from the *phase-transition region* where the probability of an instance being satisfiable is $\approx$ 50%. Randomly generated instances from the phase-transition region usually turn out to be hard for solvers such as *smodels*. It is true for problems such as 3-color problem and vertex cover problem. We observed, however that randomly generated graphs with no Hamiltonian cycle had turned out to be trivial for both *cr-smodels* and *smodels* since these random benchmarks are usually under-constrained. In this work we present two techniques: *expansion with 8-spoiler graph* and an *n-ring join* to generate hard instances of graphs with no Hamiltonian cycle.

1.6 Outline of Thesis

The remainder of the thesis is organized as follows:

Chapter2: Stable Logic Programming

In this chapter, we present stable logic programming as an important ASP formalism. We then define the stable model semantics for propositional theory and later lift the semantics for programs with variables. At the end of the chapter, we will discuss how semantics can be extended for rules with cardinality constraints.

Chapter 3: Smodels

Here, we describe in detail the implementation of the system *smodels*.

Chapter 4: New heuristics for smodels

In this chapter, we present our work in developing an alternative search heuristic to compute stable models that we call *cr-smodels*.

Chapter 5: Generating benchmarks

In this chapter, we discuss the importance of graphs with no Hamiltonian cycle and argue why these graphs are difficult to generate randomly. Then, we describe two techniques *Expansion with 8-spoiler graph* and an *n-ring join* that can generate hard instances of graphs with no Hamiltonian cycle.

Chapter 6: Experimental results

We present experimental results of comparisons between *smodels* and *cr-smodels* on various benchmarks.

Chapter 7: Discussion

In this chapter, we conclude and describe other future possible research direction in the area of answer set programming, related to our work.

Chapter 2

Stable Logic Programming

In this chapter we introduce the stable model semantics for logic programs. The stable logic programming (SLP) has syntax of DATALOG$^{\neg}$, with some extensions that are discussed later in this chapter. A language of DATALOG$^{\neg}$ is similar to the language of Horn Logic Programming with two differences:

1. presence of negation in the language.

2. absence of function symbols.

A rule in the syntax of DATALOG$^{\neg}$ is a universally quantified expression of the form [5]:

$$H \leftarrow A_1, ..., A_m, notB_1, ..., notB_n. \tag{2.1}$$

where $n \geq m \geq 0$ and each H, A_i. B_i are atoms. A logic program in SLP is a finite set of rules that represent constraints of the problem.

We first start by describing the stable model semantics for the propositional case and then lift the semantics for the programs that contains variables. At the end of the chapter, we describe how stable model semantics can be extended for the rules with cardinality constraints.

2.1 Stable Model Semantics for Propositional Logic Programs

For a propositional program P, the stable models are defined as follows.

Definition 2.1 *The reduct P^M of a propositional program P with respect to set of atoms M is the Horn program obtained from P by deleting*

1. *each rule that has the negative literal not B in its body with $B \in M$ and*
2. *all negative literals of form not B from the remaining rules.*

Example 2.1 *Consider the following program P.*

$p \leftarrow$ *not* $q,\ r$

$s \leftarrow q,\ t$

Let $M = \{q\}$. Then P^M is given by

$s \leftarrow q,\ t$

The intuitive explanation of reduct P^M is that if M is a set of ground atoms that are considered true, then any rule that depends on literal *not* b, such that $b \in M$, cannot be used in deduction and may be discarded. Also from the body of the rule if every literal *not* b, where b is not in M is trivially true and may be discarded.

In the reduct the negative body literals of the potentially applicable rules are removed. Hence, the rules are *Horn clauses* so the reduct P^M has a least model.

Definition 2.2 *M is a stable model of P if M is a least model of P^M $M = LM(P^M)$.*

Informally, if M is a set of atoms that logically follow from P^M then it is rational to believe in M.

Example 2.2 *Consider the propositional program π:*

$q(1) \leftarrow p(1,1), \neg q(1).$

$q(1) \leftarrow p(1,2), \neg q(2).$

$q(2) \leftarrow p(2,1), \neg q(1).$

$q(2) \leftarrow p(2,2), \neg q(2).$

Let $M = \{q(2)\}$. Then π^M is

$p(1,2).$

$q(1) \leftarrow p(1,1).$

$q(2) \leftarrow p(2,1).$

The least Herbrand model of the program is $\{p(1,2)\}$. It is different from M, so that M is not stable. If we try $M = \{p(1,2), q(1)\}$, then π^M is

$p(1,2).$

$q(1) \leftarrow p(1,2).$

$q(2) \leftarrow p(2,1).$

The minimum Herbrand model of this program is $\{p(1,2), q(1)\}$*, i.e.* M*. Thus,* M *is stable.*

In the next section, we discuss how the semantics of stable models can be lifted to the programs with variables.

2.2 Lifting semantics of SLP for programs with variables

The stable model semantics of the ground programs can be extended to the program with variables by employing the notion of *Herbrand Models* [11]. A *grounding* transforms a logic program P into an equivalent ground logic program $ground(P)$ (discussed in the case of Horn programs 1.2) where equivalence is defined as having the same set of stable models. A naive method of grounding uses full Herbrand instantiation of the program.

Example 2.3 *Consider the following program* π*:*

$p(1,2).$

$q(x) \leftarrow p(x,y), \neg q(y).$

Now, grounding the program we get

$q(1) \leftarrow p(1,1), \neg q(1).$

$q(1) \leftarrow p(1,2), \neg q(2).$

$q(2) \leftarrow p(2,1), \neg q(1).$

$q(2) \leftarrow p(2,2), \neg q(2).$

Definition 2.3 *(Original definition of Stable Models [17]) Let* M *be the subset of the Herbrand base of program* P*.* M *is a stable model of* P *if* M *is the stable model of ground(P)*

It means that a set of literals M is a stable model of a program P if and only if M is a stable model of the propositional theory T obtained by grounding of P.

We compute stable models of a logic program P with variables by first grounding P to its equivalent propositional theory $ground(P)$ and then we compute stable models of $ground(P)$ which by Definition 2.3 are also the stable models of P.

Now, we present the encoding of graph 3-colorability problem and Hamiltonian cycle problem in SLP to illustrate the use of SLP as an ASP systems.

Example 2.4 ***Graph 3-colorability problem:*** *Given a graph, find an assignment of one of* 3 *colors to each vertex of the graph such that vertices connected with an edge do not have same color. Below is a logic program, extracted from [23], that encodes graph 3-colorability problem.*

The data D in terms of ASP is given as

vertex(a). vertex(b). vertex(c).

edge(a,b). edge(b,c).

color(r). color(g). color(b).

The program P in terms of ASP is given by

col(X,r) $\leftarrow$ *vertex(X), not col(X; g), not col(X; b).*

col(X,b) $\leftarrow$ *vertex(X), not col(X; g), not col(X; r).*

col(X,g) $\leftarrow$ *vertex(X), not col(X; r), not col(X; b).*

$\leftarrow$ *edge(X; Y), col(X; C), col(Y; C), color(C).*

The first two lines represent encoding of a graph shown in Figure 1.3 that is triangle is shape. The third line defines the color *red, green* and *blue.*

Lines 4-6 specify that each vertex can be colored with exactly one color. The line 7, specifies the legal coloring of a graph.

Graph G has ais 3-coloring if there exists at least one stable model of P. By Definition 2.3, P has a stable model iff *ground(P)* a has stable model.

Example 2.5 *A program for* ***Hamiltonian cycle*** *problem: It is a problem of finding a path in a graph that visits each vertex of the graph exactly once and returns to the starting vertex.*

The data D is a graph G given as database of two relations:

vertex(.). *edge(.,.).*

Below is the logic program P, extracted from [23], that encodes the Hamiltonian cycle problem.

hc(V,U) $\leftarrow$ *edge(V,U), not otherroute(V,U).*

otherroute(V,U) $\leftarrow$ *edge(V,U), edge(V,W), hc(V,W),* $U \neq W$.

otherroute(V,W) $\leftarrow$ *edge(V,U), edge(W,U), hc(W,U),* $V \neq W$.

reached(U) $\leftarrow$ *edge(V,U), hc(V,U), reached(V), not initialnode(V).*

reached(U) $\leftarrow$ *edge(V,U), hc(V,U), initialnode(V).*

$\leftarrow$ *vertex(V), not reached(V).*

The first three rules ensure that for each vertex there is exactly one incoming and outgoing edge that belongs to the cycle. The last three rules state that the selected edges form a cycle that visits all vertices of a graph G. The graph G has a Hamiltonian cycle iff $ground(P)$ has a stable model.

smodels restricts the syntax to DATALOG$^{\neg}$, by eliminating function symbols from the language. If function symbols are allowed, then models can be infinite and highly complex, and the general problem of existence of a finite logic program is not even semi-decidable [25]. On the other hand, when function symbols are are not used, stable models are guaranteed to be finite and can be computed [27].

Herbrand instance of the logic program may be very large, thus computing it may be cumbersome. Since the ground instantiation of the program is always computed before computing stable models, a large ground instantiation could be a big overhead and may lead to an unacceptable performance. In order to handle this grounding problem, authors in [13] describes *domain restricted programs* which compute a subset of the ground instantiation such that this subset has exactly the same stable models, as that of whole ground instantiation.

Next we describe, how the semantics of logic programs have been extended with the new constructs of cardinality rules.

2.3 Extension of semantics

Programs written in SLP provide a framework where a variety of combinatorial, constraint satisfaction and planning problems can be handled [11]. However, there are conditions that are hard to capture using programs written in SLP. Thus, the system *smodels* was widened by extending the underlying language with more expressive constructs for representing constraints with cardinality and restrictions on them [11]. Such extensions are supported with efficient implementation techniques.

A *cardinality atom* is an expression of the form:

$$m\{p_1, ..., p_k\}n$$

where all p_i's are atoms and m and n are non-negative integers giving upper and lower bounds of the constraint, respectively. Such a constraint is satisfied by a model if the

cardinality of the subset of the literals p_1, ..., p_k satisfied by the model is between the integers m and n (inclusively). A *clause* is an expression of the form:

$$h \leftarrow a_1, \ldots, a_m$$

where each h, a_i is a cardinality atom. A logic program is a finite set of clauses.

The stable model semantics can be extended to the semantics for cardinality rules. The details of this work can be found in [12]. One consequnce of the change in the language and semantics is that stable models of the extended syntax need not to be minimal. Now we will illustrate the use of cardinality atoms with some examples. Consider an example of vertex cover problem, where the task is to find a subset of vertices of size less than k, such that for each edge(v, u) at least one of the vertex v or u is included in the cover. Now we will present, how the problem of vertex cover can be encoded using rules with cardinality constraints. For each edge(v, u) in the graph a rule

$$1\{v, u\} \tag{2.2}$$

is included and then an integrity constraint type of rule

$$\leftarrow k\{v_1, \ldots, v_n\} \tag{2.3}$$

is added where $\{v_1, \ldots, v_n\}$ is a set vertices in the graph. The first rule, expresses a choice with a cardinality saying that at least one vertex for each edge should be selected and the second rule restricts the size of cover.

Rule (2.2) is not expressible by normal rules without introducing new atoms in the program, because for normal rules stable models are subset minimal. i.e. a stable model cannot be proper subset of another stable model. However, for cardinality rules this is possible. Furthermore, there seems to be no compact encoding of the cardinality restriction shown in (2.3) using normal rules. Hence, cardinality constraints appear to be a useful extension to logic programs.

2.4 Computing Stable Models

smodels computes the stable models of the program in two phases. The first phase, implemented in the program *lparse* [21] grounds the logic program into set of ground rules. The second phase, implemented in program *smodels* uses a search heuristic to compute the

stable models of the ground theory generated by *lparse*. In the next chapter, we describe the program *smodels* in detail.

Chapter 3

Smodels

In this chapter we discuss how *smodels* computes stable models. The first section gives an overview of the algorithm. The second section describes the lookahead procedure used by *smodels* to examine the search space. The third section, describes how *smodels* picks an atom for branching.

3.1 Overview of the algorithm

To compute stable models, *smodels* uses a Davis-Putnam-like search algorithm that enumerates subsets of atoms in the program and tests each subset for stability. In the process, to compute stable models, *smodels* makes use of the properties of stable model semantics to prune the search space [6].

Let $A = \{a_1, a_2, a_3, \ldots, \textit{not}\ b_1, \textit{not}\ b_2, ...\}$. We define $A^+ = \{a \in \textit{Atoms}\text{: } a \in A\}$ and A^- $= \{a \in \textit{Atoms}\text{: } \textit{not}\ a \in A\}$.
We then define $\textit{Stable(A)} = \{M\text{: M stable model of } P \text{ and } A^+ \subset M \text{ and } A^- \cap M = \emptyset\}$,

It follows that for an atom a, if both a and *not* a appear in A, then the set of stable models that A represents is empty. The *smodels* algorithm begins with A as an empty set, it then adds literals to A and checks whether the resulting set corresponds to at least one stable model. If it does not, then it backtracks by replacing recently assigned literal to A by its complement (i.e. replacing literal a by *not* a and vice versa). If $\textit{Stable}(A) \neq \emptyset$, then we call the set A *partial stable model* since A can be extended to a stable model.

The search space consisting of all possible configurations of A is pruned by deducing additions to A from the program using properties of stable model semantics. For example, if the rule

$$a \leftarrow b,\ not\ c$$

is in a program, then

1. if (b, *not* c) ϵ A, then we deduce that every stable model must contain a and add a to A.

2. if (b, *not* a) ϵ A, then we deduce that every stable model in *Stable(A)* must contain c and add a to A.

3. if this is the only rule with a in the head and a ϵ A, then every stable model in *Stable(A)* contains b, *not* c and we add b, *not* c to A.

These type of deductions form a stable model counterpart to *unit propagation.* If for some atom a, both literals a and *not* a belongs to A, then we say that a *conflict* or a *contradiction* is found and the algorithm backtracks.

Unit propagation can be strengthened by the lookahead - a method to dicover the only possible truth assignment to literal based on the fact that the opposite assignment leads to a direct contradiction in the unit propagation. When no more truth assignments can be deduced through unit propagation and look-ahead, *smodels* picks an atom (procedure *heuristic*) for branching.

The algorithm presented below summarizes the decision process of *smodels.* The function *smodels(P,A)* in the algorithm returns true if there is a stable model of the program P agreeing with the set of literals in A.

function *smodels*(P, A)
A := *expand*(P,A)
A := *lookahead*(P,A)
if *conflict*(P, A) **then**
  return false
else if A covers *Atoms(P)* **then**
  return true
else

```
    x := heuristic(P,A)
    if smodels(P, A ∪ {x}) then
      return true
    else
      return smodels(P, A ∪ {not x})
    end if
end if
```

The subroutine *lookahead* performs look-ahead to identify literals that immediately give rise to conflicts. Function *heuristic* identifies the literal for branching. The implementational details of the two functions *lookahead* and *heuristic* are discussed later in this chapter.

The function *expand* uses *unit propagation* to discover new atoms that can be added to partial model A. A unit propagation consists of unit resolution and unit subsumption. Unit resolution removes all literals that are false from every clause and unit subsumption removes all clauses that contain a literal that is true [18].

The function *expand(P, A)* returns set A', where A' consists of literals in set A and newly discovered atoms during the unit propagation. The function *expand* maintains the semantics of stable model since it satisfies the following conditions [23]:

1. $A \subseteq A'$, and

2. every stable model of P that agrees with A also agrees with A'.

A good implementation of function *expand* is very important because *expand* is often called in *smodels* and on each call *expand* reduces the search space by determining new atoms.

The function *conflict* returns true on a contradiction. Now, we will discuss the procedure *lookahead* in detail.

3.2 lookahead

The idea behind *lookahead* procedure is to prune the search space by determining the literals that instantly give rise to conflict. *lookahead* identifies these literals by examining all the literals that are not in A. We refer this set of literals as B.

```
function lookahead(P, A)
repeat
  A' := A
  A := lookahead_once(P, A)
  until A = A'
return A

function lookahead_once(P, A)
  B := Atoms(P) - Atoms(A)
  B := B ∪ not B
 while B ≠ φ do
   Take any literal x ϵ B
   A' := expand(P, A ∪ {x})
   B := B - A'
   if conflict(P, A') then
     return expand(P, A ∪ {not (x })
   end if
 end while
 return                                                   A
```

Figure 3.1: Algorithm for lookahead

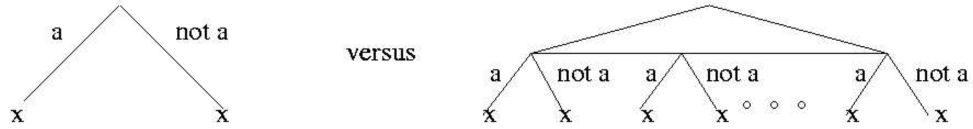

Figure 3.2: Reduction of search space with lookahead

For each literal x in B, *lookahead* (shown in Figure 3.1) calls $expand(P, A \cup \{x\})$ and $expand(P, A \cup not\ \{x\})$, and then satisfies one of the following three conditions:

1. If both $(A \cup \{x\})$ and $(A \cup \{not\ x\})$ raise a conflict, then *lookahead* finds a contradiction and returns false.

2. If none of $(A \cup \{x\})$ and $(A \cup \{not\ x\})$ raises a conflict, then no important information is gathered and *lookahead* continues with next literal in B.

3. If $(A \cup \{x\})$ give rise to a conflict and $(A \cup \{not\ x\})$ does not, then *not* x should be in A. On adding *not* x to A, the function *expand* is called with argument $(A \cup \{not\ x\})$ to determine new literals that will reduce the search space. It is similar when $(A \cup not\ x)$ give rise to a conflict.

The Figure 3.2 depicts how the search space is reduced by the *lookahead*.

3.3 Heuristic

The function *heuristic* is called on all atoms identified during *lookahead* that do not produce a conflict. From this set of atoms, we need to pick one atom for branching that will minimize the remaining search space. The function *heuristic* helps in picking such an atom for branching.

The choices made by a heuristic can drastically affect the time the algorithm has to spend in search for a solution [23]. A correct choice brings the algorithm closer to a solution while a wrong choice will lead the algorithm astray.

For a atom x, let

$$A_p = expand(P, A \cup \{x\})$$

and

$$A_n = expand(P, A \cup \{not\ x\}).$$

The *positive weight* p of an atom x is the number of new literals determined by *expand*$(P, \{A \cup x\})$ and the *negative weight* n of x is the number of new literals determined by *expand*$(P, \{A\cup$ *not* $x\})$. Therefore, $p = |A_p$ - $A|$ and $n = |A_n$ - $A|$.

Assume that search space is a full binary tree of height H. A full binary tree is a binary tree whose paths from roots to the leaves are of equal length. Then,

$$2^{H-p} + 2^{H-n} = 2^{H}\frac{2^n+2^p}{2^{p+n}}$$

is an upper bound of the size of remaining search space [23]. Minimizing this number is equivalent to minimizing

$$\log\frac{2^n+2^p}{2^{p+n}} = \log(2^n + 2^p) \text{ - (p+n)}.$$

Since

$$2^{max(n,p)} < 2^n + 2^p \leq 2^{max(n,p)+1}$$

is equivalent to

$$\max(n,p) < \log(2^p + 2^n) \leq \max(n,p) + 1$$

and, consequently, to

$$\text{-}\min(n,p) < \log(2^n + 2^p) \text{ - } (p+n) \leq 1 \text{ - } \min(n,p),$$

it suffices to maximize $\min(n,p)$. If two different literals have equal minimum, then *smodels* chooses an atom with greater maximum, as this minimizes $2^{-max(n,p)}$. When the best literal x has been found, then heuristic returns x or *not* x (based on which of them will reduce search space the most).

3.4 Problem Statement

Our goal in this work is to develop a better search heuristic that we call *cr-smodels*. *cr-smodels* wisely picks an atom for branching that is expected to minimize the size of the

remaining search space. The heuristic is aimed to give better performance on a wide range of benchmark problems.

We compare the performance of *cr-smodels* with *smodels* on the randomly generated benchmark instances. We found that often randomly generated graphs with no Hamiltonian cycle turned out to be easy for both *cr-smodels* and *smodels*. A Hamiltonian cycle problem being an important benchmark problem, our other goal is to design techniques that generates hard instances of graphs with no Hamiltonian cycle.

Chapter 4

New Heuristic

In this chapter we present a new search heuristic that we call *cr-smodels*, the resulting version of smodels. We first describe, how *cr-smodels* estimates the size of the search space by solution to a linear recurrence relation. Then we explain, how *cr-smodels* computes the size of the remaining search space and picks up an atom for branching that is expected to minimize the size of the remaining search space.

4.1 Estimation of search space

Consider the set B which consists of all atoms x such that neither $(A \cup \{x\})$ nor $(A \cup \{not\}\{x\})$ yield a conflict in the *lookahead* process. The function $estimate(n)$, estimates the size of the remaining search space, when there are n unassigned literals left (that is, there are n literals in set B). For each atom x, such that $x \in B$, let p_x and n_x be its positive and negative weights, respectively. On adding x to the partial set A, there are n - p_x atoms in set B. Thus, we estimate the remaining search space by $estimate(n - p_x)$, similarly on adding literal *not* x to A search space is $estimate(n - n_x)$.

Figure 4.1 is the representation of the part of the search space, where the parent node estimates the search space as $estimate(n)$ and the left and right child nodes estimate the search space on adding x and *not* x to A, respectively. If x is used for branching, then

$$estimate(n) = estimate(n - p_x) + estimate(n - n_x) \tag{4.1}$$

Similarly, we obtain a linear recurrence for remaining atoms in B. The idea is to pick an atom x, such that this recurrence has lowest possible growth rate. The estimate of size

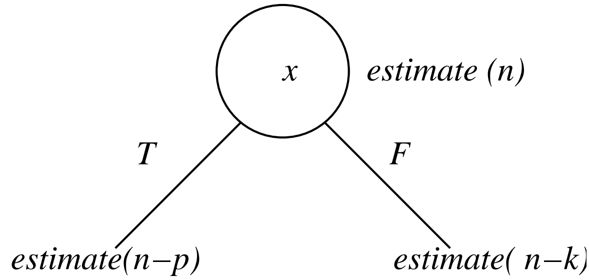

Figure 4.1: Estimation of search space.

of the remaining search space is low, if the linear recurrence has lower growth rate. The growth rate of the sequence, defined by a linear recurrence, is given as the root of the characteristic of the recurrence relation. In case of the recurrence (4.1), this characteristic equation is

$$X^{p_x} = 1 + X^{p_x - n_x}$$

To solve it, we need to find a root of the function

$$f(X) = X^{p_x} - X^{p_x - n_x} - 1 \quad (4.2)$$

4.2 Computing roots of the equation

The graph of the function (4.2) is negative at $x = 1$ and positive at $x = 2$. Since function (4.2) has positive first derivative in the interval [1,2], it has exactly one root in this interval. We use Newton-Raphson method to compute this root. The linear relation in (4.1) converges rapidly on starting with point $x = 2$.

Each iteration of the Newton Raphson method leads us closer to the desired roots as shown in figure 4.2 and converges rapidly. In our implementation, we performed iterations of Newton-Raphson until we get the error bound of $O(10^{-14})$ i.e. $\|f(x)\| < 10^{-14}$. On running *cr-smodels* on different benchmarks we found that often (≈90%) Newton-Raphson method converged in less than five iterations and the whole process was fast.

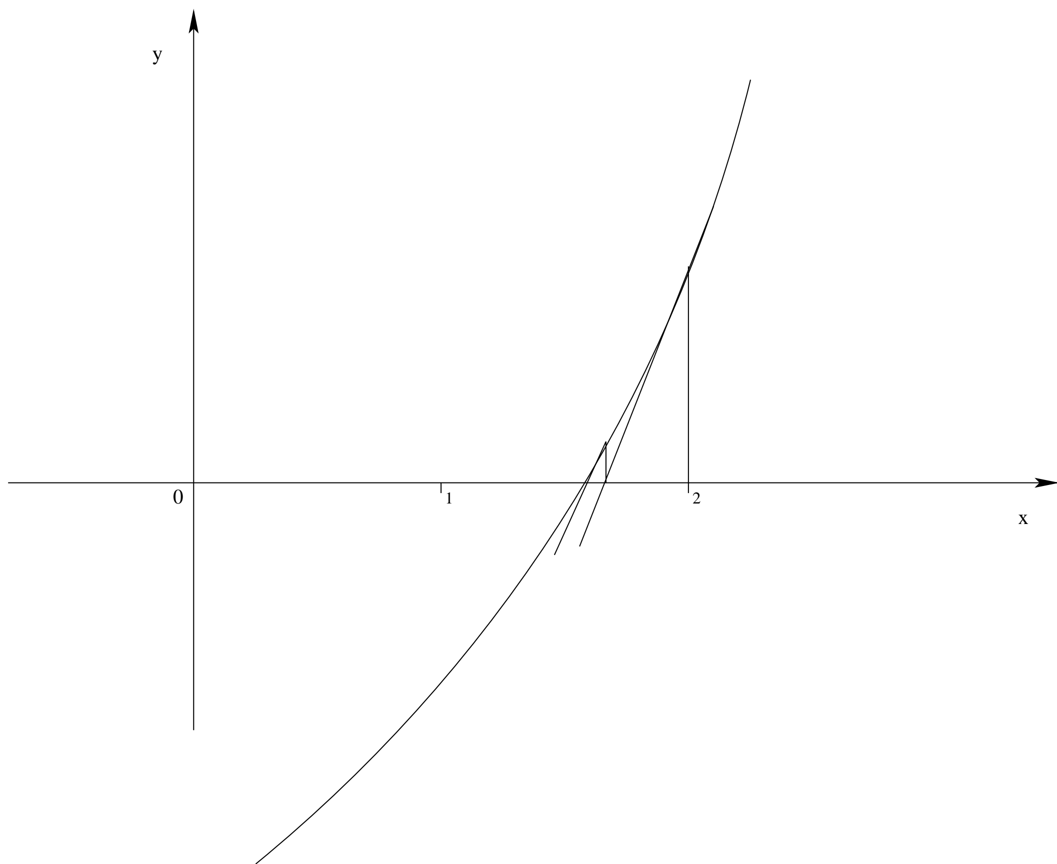

Figure 4.2: Newton-Raphson Method.

Chapter 5

Generating hard unsat HC instances

The first step to design programs for testing algorithms to compute stable models is to pick a search problem. Next, we identify parameters in the problem that can be varied when generating instances. An instance of a search problem is called a test case (or a benchmark). For example, consider the problem of vertex cover, where we choose the size of cover, number of edges and vertices in the graph as parameters to generate its benchmarks. For each problem, its benchmarks may be generated with a particular instance generator. The instance generator accepts as input, the parameters of the problem, number of benchmarks to be generated and a range that specifies the approximate probability of each generated benchmark to be satisfiable. The instance generator randomly generates benchmarks, so that there is a high probability that generated benchmarks are different from each other.

One interesting property of randomly generated benchmarks for graph problems is the occurrence of *phase transition* phenomenon [14],[15] i.e. a rapid change in satisfiability which can be observed on systematically increasing (or decreasing) the number of edges e for fixed number of vertices v in a graph. If in a problem, for small e, almost all instances are satisfiable; then at some critical $e = e'$ the probability of generating satisfiable instance drops sharply to almost zero. Beyond e' almost all instances are unsatisfiable. Intuitively, e' characterizes the transition between a region of underconstrained instances which are almost always satisfiable, to overconstrained instances which are almost always unsatisfiable. The

region characterized by e' is called a *hard region*.

To compare the performance of a heuristic it is necessary to test it on benchmarks that are *hard* to solve and have diverse properties. Our aim is to generate benchmarks that belong to the hard region and are also hard to solve (A benchmark is said to be *hard* for a heuristic H, if H often backtracks to determine the solution). We choose benchmarks that have $\approx 50\%$ probability of being satisfiable because of two reasons: first, these instances are often hard to solve and second, they are equally SAT and UNSAT.

We observed that often randomly generated satisfiable benchmarks are hard to solve. For many problems it is also the case for randomly generated unsatisfiable benchmarks. However, we observed that it is not true for Hamiltonian cycle problem. The randomly generated satisfiable benchmarks are hard to solve while randomly generated unsatisfiable benchmarks of Hamiltonian cycle problem were easy for *smodels* and *cr-smodels* to solve, that is these heuristics reached solution without involving any backtracking.

We start with defining the problem of Hamiltonian cycle in a graph.

Definition 5.1 *A Hamiltonian cycle is a sequence of vertices such that every two consecutive vertices are connected with an edge and there is an edge between the first and the last vertex.*

Since, the Hamiltonian cycle problem is important, we have put our effort in determining techniques to generate hard instances of graph with no Hamiltonian cycle. All the graphs discussed in this chapter are undirected.

The chapter is arranged in two sections, each describing a technique to generate hard instances of non-Hamiltonian graphs.

5.1 Expanding with 8-spoiler graph

In this section, we present a technique to generate graphs with no Hamiltonian cycle by expanding a graph G with *8-spoiler graph* [4] shown in Figure 5.1.

The 8-spoiler graph is an undirected 3-connected graph with 8 vertices and 16 edges. Let D be a *8-spoiler graph.* The vertices $\{a, b, c, d\}$ and $\{e, f, g, h\}$ are called its *conjunct* and *non conjunct* vertices, respectively.

Let G be any undirected random graph. A *spoiler expansion of* G is a graph obtained by joining four vertices (picked randomly) in G with the conjunct vertices of D with exactly

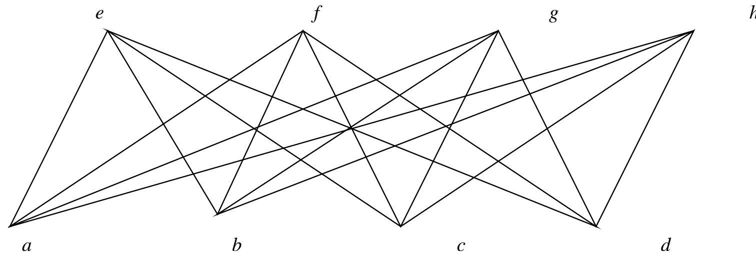

Figure 5.1: 8-Spoiler graph

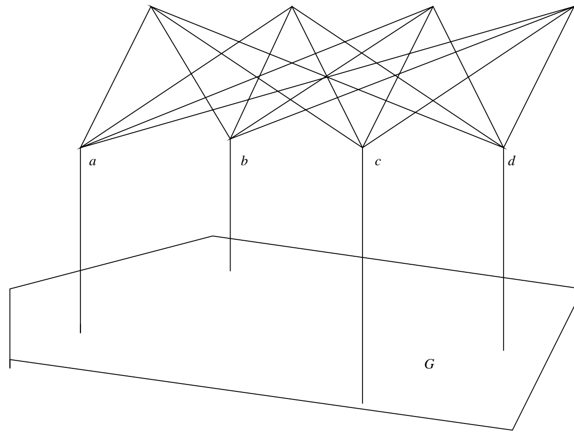

Figure 5.2: Spoiler expansion of graph G

one edge from each vertex as shown in Figure 5.2.

Theorem 5.1 *Let H be the spoiler expansion of G. Then H does not have a Hamiltonian cycle.*

Proof: Let C be a Hamiltonian cycle in H. Then C will enter graph D either once or twice.

If C enters D exactly once, then C can pass through at most three non-conjunct vertices of D. If C enters D twice, then C passes through exactly two non-conjunct vertices of D. Thus, C does not passes through at least one non-conjunct vertex of D, a contradiction $\Box$.

We used *smodels* to test the hardness of non-Hamiltonian graphs generated using technique *expanding with 8-spoiler graph.* The graphs used in experiments were randomly gen-

erated (using algorithm 6.1) and the conjunct vertices used for joining with the 8-spoiler graph were also picked randomly.

In our experiments, we generated hundred instances of random graphs for each pair of vertex and edge. We varied the number of vertices = {10,20,30} and for each vertex we varied the number of edges from { 30,40, ..., 80} in our experiments. We observed that for all the generated instances *smodels* found these graphs to be hard to solve.

5.2 Ring Join

In this section we will present another technique, that we call *n-ring join*, that generates hard instances of non-Hamiltonian graphs. n-ring join starts with picking a set of n graphs out of which at least one graph is non-Hamiltonian. Then, it joins them in a ring to obtain a larger graph that is also non-Hamiltonian. Let $G_1, G_2, ..., G_n$ be the set of n graphs picked by n-ring join. These graphs were joined in a ring by :

1. selecting an edge e(u_i, v_i) from each graph G_i,
2. then, joining all the vertices $u_1, ..., u_n$ in a cycle as shown in figure 5.3,
3. similarly, joining all the vertices $v_1, ..., v_n$ in a cycle,
4. and removing all the edges e(u_i, v_i) from their corrosponding graphs.

Theorem 5.2 *If graph H is a n-ring join of graphs G_1, G_2, ..., G_n, then H is non Hamiltonian if at least one of the graph in G_1, G_2, ..., G_n is non-Hamiltonian.*

Proof: Consider the Figure 5.3. Let H be the n-ring join of graphs G_1, G_2, ..., G_n. Let, G_1 be a non-Hamiltonian graph and C be some Hamiltonian cycle in H. C must enter and leave G_1 - $e(u_1, v_1)$ through v_1 and u_1 (say enters through u_1 and leave through v_1 or vice versa.). Moreover, it passes all vertices of G_1 - $e(v_1, u_1)$. Thus G_1 is Hamiltonian, a contradiction.

□

We used *smodels* to test the ability of n-ring join technique to generate hard non-Hamiltonian graphs. A graph G was ring joined by

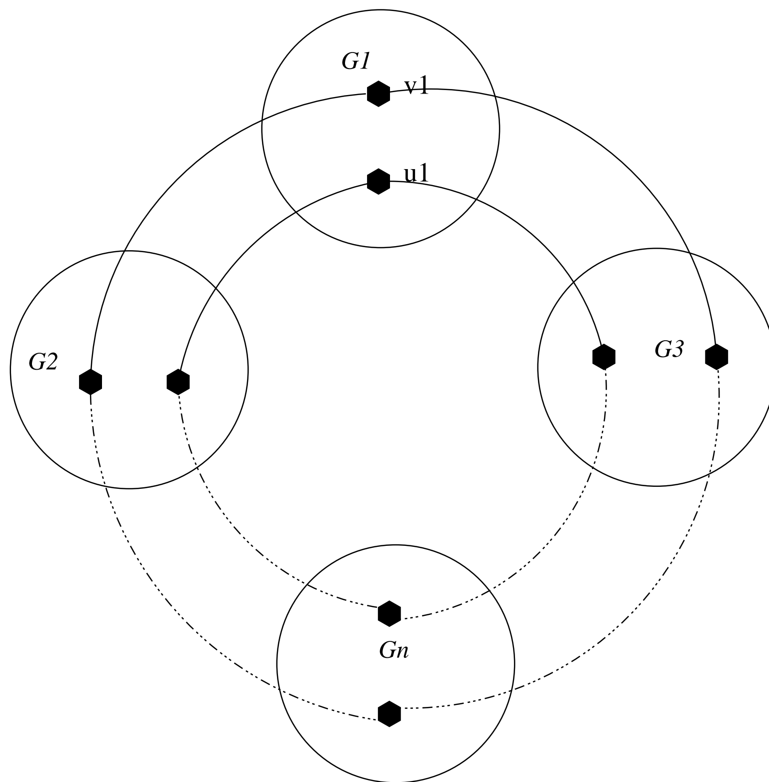

Figure 5.3: n-ring join of a graph G

1. taking n copies of G,

2. then, randomly picking an edge e in G and

3. performing n ring join of all the copies of G with their corrosponding edge of e.

Each experiment was conducted by using a graph G and varying n from 2, ..., 10. For each n hundred instances were generated. We used graphs shown in 5.3, extracted from [2], to perform ring join. We observed that these graphs were non-Hamiltonian and hard for *smodels* to solve.

5.3 Graphs used for n-ring join

The graphs shown below were taken from the work of R. E. L. Aldred, S. Bau, D. A. Holton and D. McKay in [2]. We used these graphs to generate hard unsat benchmark instances of Hamiltonian cycle problem using n-ring join problem.

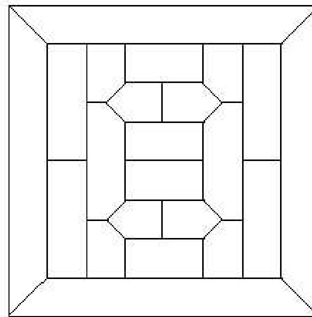

Figure 5.4: 42-vertex nonhamiltonian graph

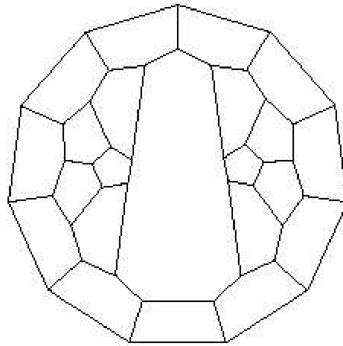

Figure 5.5: 42-vertex nonhamiltonian graph

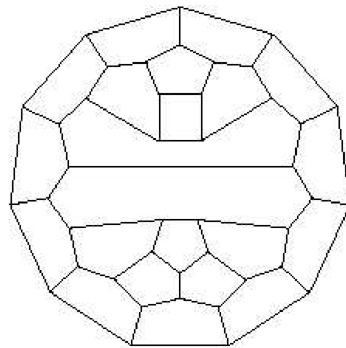

Figure 5.6: 42-vertex nonhamiltonian graph

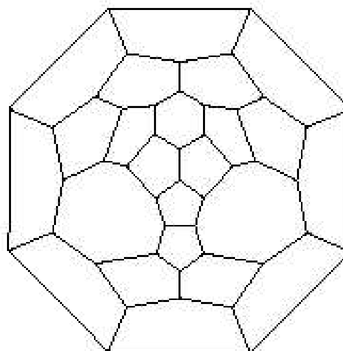

Figure 5.7: 44-vertex nonhamiltonian graph

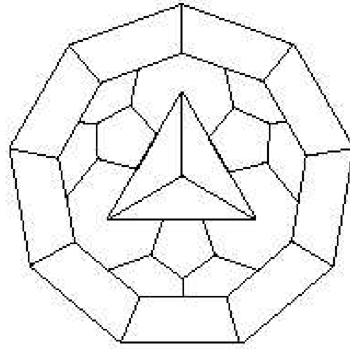

Figure 5.8: 46-vertex nonhamiltonian graph

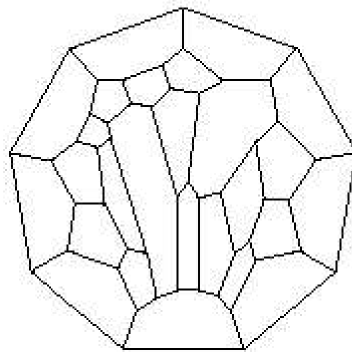

Figure 5.9: 50-vertex nonhamiltonian graph

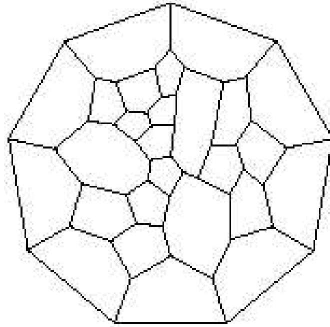

Figure 5.10: 50-vertex nonhamiltonian graph

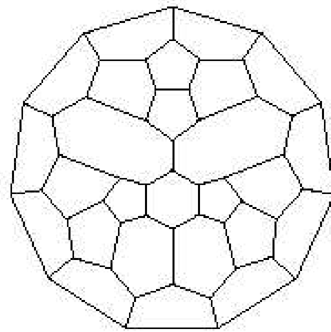

Figure 5.11: 50-vertex nonhamiltonian graph

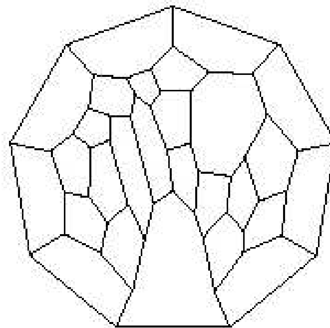

Figure 5.12: 52-vertex nonhamiltonian graph

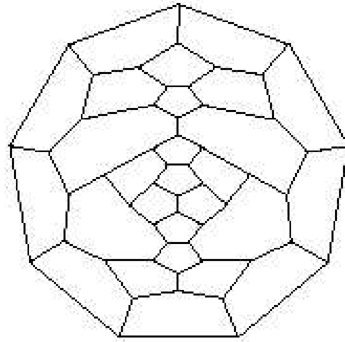

Figure 5.13: 52-vertex nonhamiltonian graph

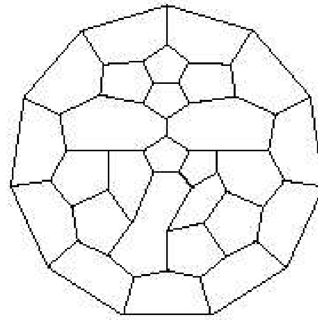

Figure 5.14: 52-vertex nonhamiltonian graph

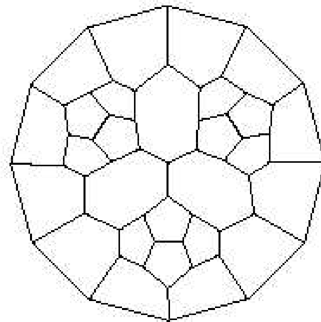

Figure 5.15: 52-vertex nonhamiltonian graph

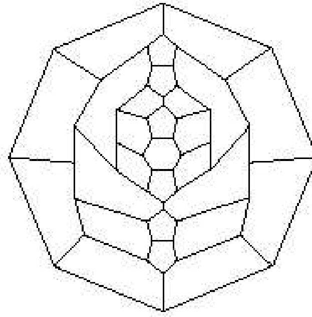

Figure 5.16: 52-vertex nonhamiltonian graph

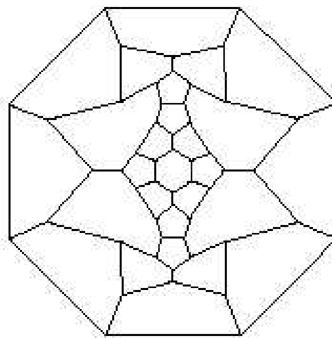

Figure 5.17: 52-vertex nonhamiltonian graph

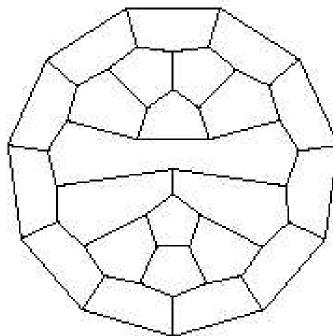

Figure 5.18: 44-vertex nonhamiltonian graph

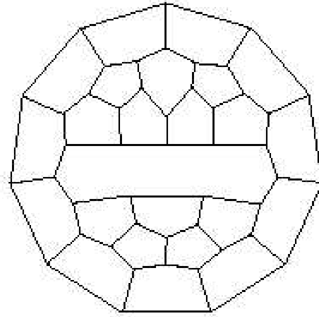

Figure 5.19: 46-vertex nonhamiltonian graph

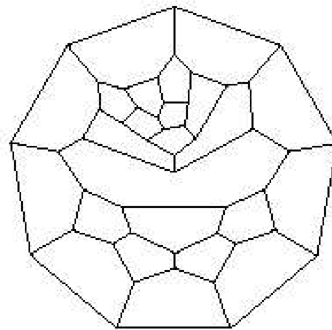

Figure 5.20: 46-vertex nonhamiltonian graph

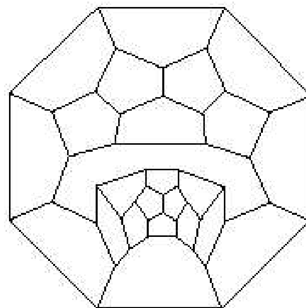

Figure 5.21: 48-vertex nonhamiltonian graph

Chapter 6

Experimentation

One of the criteria in comparing the performance of a search heuristic is the number of truth assignments obtained to reach the solution. *smodels* prints the number of truth assignments made to solve a problem. The authors [13] claims that the number of truth assignments is a good estimate of the size of the search space traversed. Our experiments confirmed the claim and had show that this number relates well with the running time of the heuristic. A heuristic efficiently solves a problem if it makes few truth assignments to reach to its solution.

In order to test a heuristic, we compare it on a wide range of problems as well as on several instances of the same problem. Instances of the same problem may vary in size and difficulty. Additionally, the heuristics that work well for some problems may work poorly for other. While comparing the performance of two heuristics, we say that a heuristic is better then the other if it is faster on large number of problems.

The logic programs used for experimentation were written in the syntax of stable logic programming. They are presented in Appendix A. We have attempted to write the programs that are as efficient as possible.

6.1 Smodels

The heuristic *cr-smodels* was implemented by modifying *smodels*[10] (implemented by Ilkka Niemelä and Patrik Simons). In this chapter, we present the performance comparision of *cr-smodels* and *smodels* on various benchmark problems. All experiments are conducted on a computer with Pentium 4 processor with 1 GB of RAM running under Linux operating

Input: The number of vertices v and the number of edges e.

Output: Graph G.

random-graph(v, e)

Set $E = \phi$

For $i = 1$ to e **do**

Randomly generate (w, u) such that edge (w, u) does not exists.

if graph is undirected then $(w, u) = (u, w)$

Add (w, u) to E

End For

Return G

End random-graph

Figure 6.1: A algorithm to generate random graphs

system.

6.2 Generating random graphs

The benchmark instances used for experimentation were generated randomly. Figure 6.1 outlines the routine written by Dr. Raphael Finkel that we used to generate random graphs. The routine accepts as input the number of vertices and edges of the graph to be generated. We used undirected graphs for vertex cover, graph 3-colorability and Hamiltonian cycle problem.

6.3 Results for selected problems

The problems used for comparison of heuristics are search problems based on graphs and combinatorial configurations. These problems are in the class NP and some of them are known to be *NP-complete*. Below we will present experimental results of comparison between *cr-smodels* and *smodels* on the following problems: Hamiltonian cycle, graph 3-Coloring, n-queen, n-queen.x and vertex cover. We will start with the Hamiltonian cycle

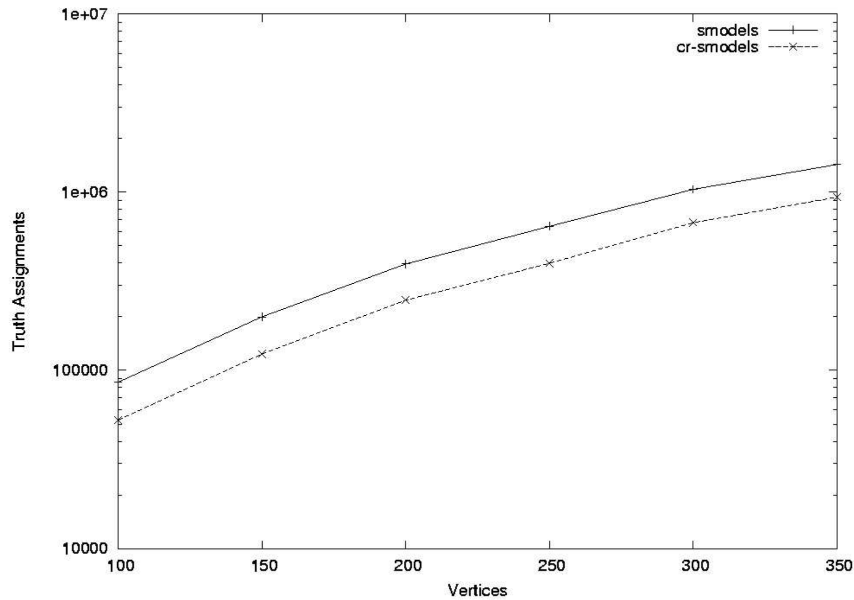

Figure 6.2: Comparison of *smodels* and *cr-smodels* on Hamiltonian cycle problem

problem.

6.3.1 Hamiltonian cycle

The Hamiltonian cycle problem is a search problem of finding a cycle such that the cycle passes through every vertex in the graph exactly once. Not only is the Hamiltonian cycle problem in itself of interest to us but many other problems like routing, planning, etc, can be encoded as a Hamiltonian cycle (or path) problem.

The Hamiltonian cycle problem demonstrates the effectiveness of a capability of a logic program to express concisely and effectively the concept of transitive closure (Chapter 5). Thus, it is an important benchmark to test a heuristic.

For our experimentation, we used graphs with number of vertices and edges = {(100, 594), (150, 842), (200, 1183), (250, 1502), (300, 1902), (350, 2248)}. For each pair of vertices and edges we generated hundred benchmarks. These benchmarks have ≈50% probability of being satisfiable. These instances are neither underconstrained nor overconstrained, therefore they belonged to the hard region.

We observed that both *cr-smodels* and *smodels* solved almost all the unsatisfiable

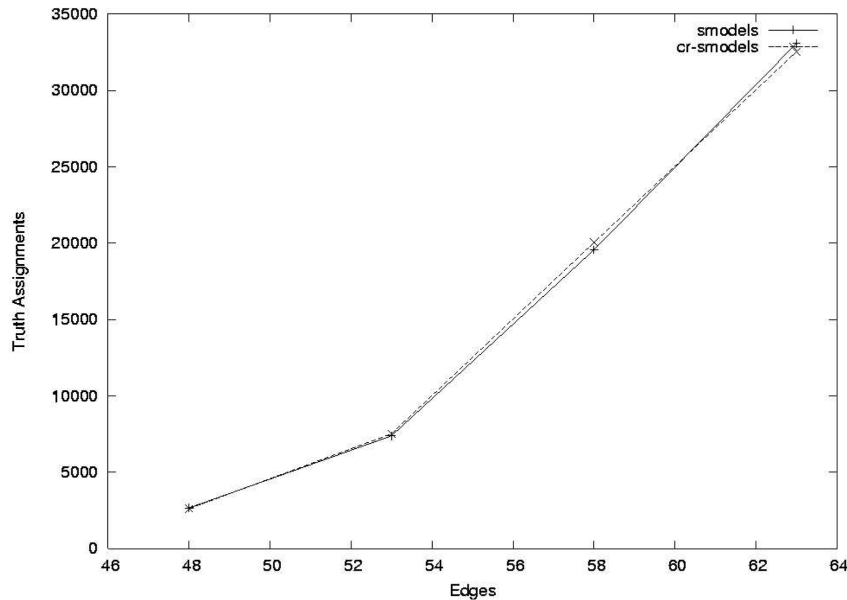

Figure 6.3: Comparison with non-Hamiltonian graphs generated using "Expanding with 8-spoiler" technique.

instances of Hamiltonian cycle without involving any backtracking. Figure 6.2 shows the comparison between *cr-smodels* and *smodels* only on the satisfiable instances. Experimental results shows that, *cr-smodels* was more efficient than *smodels*. Therefore *cr-smodels* is effective in pruning the search space and choosing the branching atoms.

To test the heuristics on unsatisfiable instances of Hamiltonian cycle, we generated non-Hamiltonian graphs using technique of *expanding with 8-spoiler graph* (discussed in Chapter 5.2). All the graphs used for experimentation had eighteen vertices but we varied the number of edges from $\{46, 48, ..., 64\}$. For each pair of vertices and edges we generated hundred benchmarks. We observed that none of the heuristics consistently outperformed the other therefore none of them could be declared as a winner. Figure 6.3 shows the result of comparision of *smodels* and *cr-smodels*.

Graphs shown in Appendix B were used to generate another set of non-Hamiltonian graphs, using *n-ring join* technique. For each n, we generated a hundred instances of graphs (as described in 5.2). In our experiments we varied n from $\{2, ..., 10\}$.

Figure 6.4 presents the result of performance of *cr-smodels* and *smodels* on benchmarks

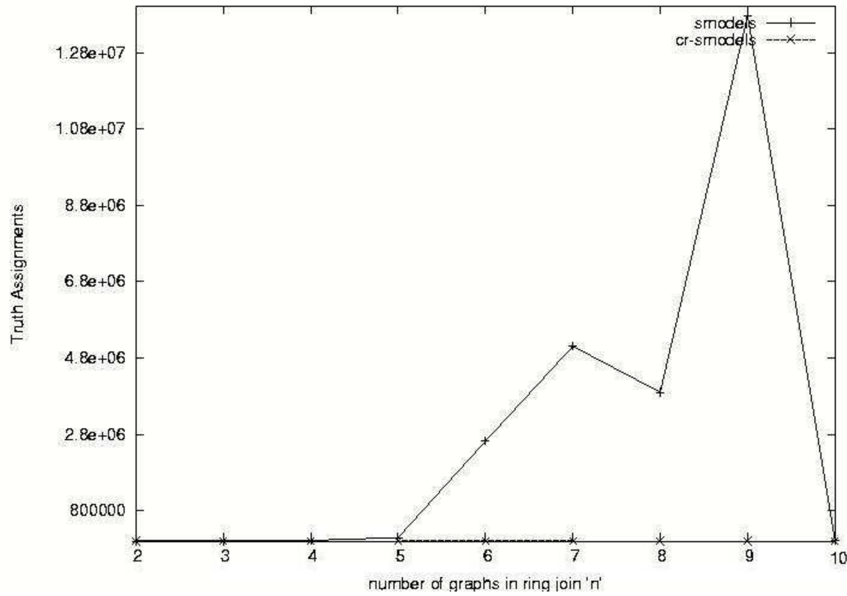

Figure 6.4: Comparison with non-Hamiltonian graphs generated using "n-ring join" technique.

generated with n-ring join using graph 5.3. *cr-smodels* outperformed *smodels* on all the instances. We observed that, in the time frame of six hours and for $n > 5$, *smodels* was not able to reach a solution for the benchmark instances generated from graphs 5.3, 5.3, 5.3, 5.3.

6.3.2 Graph 3-Coloring

Graph 3-coloring is a problem of assigning exactly one color to each vertex of a graph such that vertices joined by an edge are not assigned the same color.

We experimented on graphs with {(75,162), (100,225), (125,293), (150, 346), (175,395), (200,458), (225,518)} number of vertices and edges. For each pair of vertices and edges, we generated hundred benchmark instances. All the instances have probability of being ≈50% satisfiable, thus they belonged to hard region.

Figure 6.5 presents comparision of *cr-smodels* and *smodels* on graph 3-color problem. We observed that the performance of *smodels* was slightly better than *cr-smodels*.

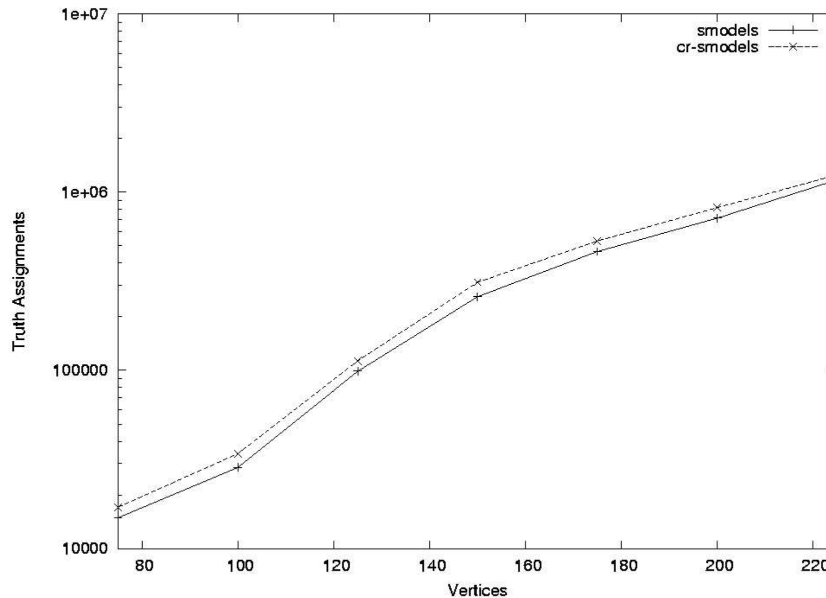

Figure 6.5: Comparison of *smodels* and *cr-smodels* on graph 3-coloring problem

6.3.3 N-Queen

The n-queen problem is a problem of placing n queen on a $n \times n$ chess board such that no two queens attack each other i.e. two queens cannot be placed in same row, column and diagonal. A n-queen problem may be satisfiable for $n > 3$. A n-queen problem may have many solutions but in our experiments, we look for the first solution that is determined. Figure 6.6, shows the result of comparison of *cr-smodels* and *smodels* by varying n from $\{18$ to $25\}$.

We observed that, *cr-smodels* was more efficient than *smodels* on most of the instances with $n = 23$ and 24 as exceptions.

6.3.4 n-queen.x

The *n-queen.x* problem is an extension of *n-queen*. The problem is to find the placement of $n - x$ queens on $n \times n$ chess board after x queens has been pre-assigned. The pre-assignment of queens is selected randomly using a program such that pre-assigned queens does not attack each other.

The experiments were conducted by varying n from $\{17$ to $30\}$. For each n, we varied

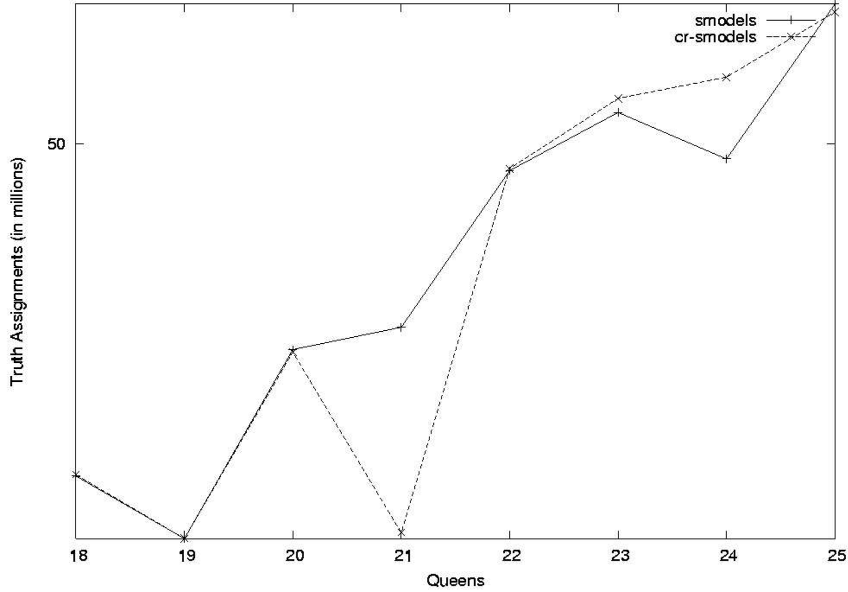

Figure 6.6: Comparison of *smodels* and *cr-smodels* on N-queen problem

x from $\{2$ to $9\}$ and for each x, we generated hundred benchmark instances.

Figure 6.7 presents the comparision of *cr-smodels* with *smodels* on n-queen.x problem for n = 24. Result shows that, *cr-smodels* averaged better than *smodels* on most of the instances. *cr-smodels* was more efficient than *smodels* in the hard region, which varied from $x = \{4, ..., 7\}$ in our experiments.

6.3.5 Vertex Cover

For a given $G = (V, E)$, the problem of vertex cover is to find the smallest subset $S \epsilon V$ such that for each edge $u, v \epsilon E$ at least on of u or v are in S.

We experimented on graphs with {(80,138,40), (85,149,42), (90,161,45), (95, 167,47), (100, 160,48),(105,174,53),(110,181,56),(115,189,61), (120, 197,64)} number of vertices, edges and size of cover. For each set of vertices, edges and size of cover we generated hundred benchmark instances such that they have ≈50% probability of being satisfiable. These instances belongs to hard region.

We observed that for in the experiments *cr-smodels* and *smodels* made exactly same number of truth-assignments.

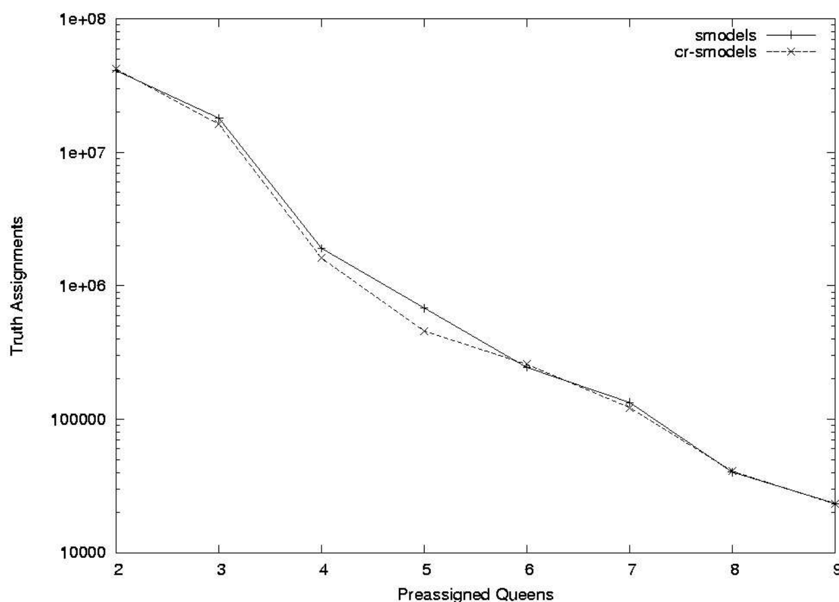

Figure 6.7: Comparison of *smodels* and *cr-smodels* on n-Queen.x problem

6.4 Summary of Results

We experiment in many cases more with problems based on graphs and combinatorics. The results shows that *cr-smodels* is efficient (with respect to number of truth assignments made) than *smodels*. *smodels* is the state of art implementation that has been under development and in use for several years.

In all the instances of Hamiltonian problem, *cr-smodels* performs better than *smodels*. In non-Hamiltonian graphs generated by technique *n-ring join*, *cr-smodels* noticeably outperformed *smodels*, while none of them could be declared as winner on graphs generated with technique *expanding with 8-spoiler graph* technique. In N-queen and N-queen.x problems *cr-smodels* is efficient on most of the instances. On n-queen.x problem in the hard region $cr - smodels$ outperformed *smodels*. Graph 3-color problem is the only example where *smodels* notably outperformed *cr-smodels*.

The results of the experiment show that in most of the cases *cr-smodels* is more efficient than *smodels*. We believe that *cr-smodels* has applicability to solve a large range of practical problems.

Chapter 7

Discussion

Our primary goal in this work was to present the new search program *cr-smodels* that computes stable models. The heuristic used by *cr-smodels* tries to reach the solution of a problem with minimum search space traversal.

Experimental results show that *cr-smodels* notably outperformed *smodels* in solving Hamiltonian cycle problem, n-queen problem and unsatisfiable instances of Hamiltonian cycle generated using n-ring join technique. For n-queen.x and vertex cover problems the performance of *cr-smodels* was comparable to *smodels*. Therefore, we believe that *cr-smodels* may be a viable alternative to programs like *smodels*.

cr-smodels has applicability to solve large number of problems. However, there are several issues that requires further investigation. First, the heuristic *cr-smodels* can be improved to enhance its performance especially on problems like graph coloring, where *cr-smodels* was often notably outperformed by *smodels*. Second, there is a need to test *cr-smodels* on a wider range of problems.

Our other goal was to present techniques, like *expanding with 8-spoiler graph* and *n-ring join* (discussed in chapter 5), to generate graphs with no Hamiltonian cycle. Experimental results shows that both *smodels* and *cr-smodels* found these graphs *hard* to solve. This work can be extended further to develop techniques for generating hard unsatisfiable instances of problems like graph coloring, n-queen.x, since randomly generated unsatisfiable instances of these problems are often trivial to solve.

Appendix A

Logic Programs

We present the logic programs that we used for experimentation. The problems are encoded in Stable Logic Semantics.

A.1 Hamiltonian Cycle problem

Program to compute a hamiltonian cycle in a directed graph. The path is to start in a specified vertex, defined by init_vtx predicate.

```
% For each vertex there is exactly one edge leaving it.
1 {inhm(X,Y): edge(X,Y) }1 :- vtx(X).
% For each vertex there is exactly one edge entering it.
1 {inhm(X,Y): edge(X,Y) }1 :- vtx(Y).
% All vertices must be reached from the initial vertex
reached(Y) :- vtx(Y), init_vtx(X), inhm(X,Y).
reached(Y):- edge(X,Y), inhm(X,Y), reached(X).
:- vtx(X), not reached(X).
init_vtx(0).
```

A.2 Graph 3-color

The problem of 3-coloring of a graph is that of assigning 3 colors to vertices such that each vertices is assigned exactly one color and vertices joined by an edge are not assigned the same color.

hide.

show col(X, Y).

% Defining valid colors.

color(red).

color(blue).

color(yellow).

% A vertex can be colored with atmost one color.

col(X,red) :- vtx(X), not col(X, blue), not col(X,yellow).

col(X,blue) :- vtx(X), not col(X, red), not col(X,yellow).

col(X,yellow) :- vtx(X), not col(X, blue), not col(X,red).

% Two vertex with a common edge cannot be colored same.

:- edge(X,Y), color(C), col(X,C), col(Y,C).

A.3 N-queen

Domain predicate "d" specifies the number of queens and the dimensions of the board. The main predicate is the predicate "q". Atom "q(X,Y)" represents the fact that there is a queen in the position (X,Y).

% Domain predicate: the dimension of the board.

d(1..n).

% Exactly one queen in each row

1 { q(X,Y):d(Y) } 1 :- d(X).

% Exactly one queen in each column

1 { q(X,Y):d(X) } 1 :- d(Y).

% Exactly one queen on each diagonal

:- d(X;Y;X1;Y1), q(X,Y), q(X1,Y1), X ¡ X1, Y != Y1, abs(X - X1) == abs(Y - Y1).

A.4 N-queen.x

Domain predicate "d" specifies the number of queens and the dimensions of the board. The main predicate is the predicate "q". Atom "q(X,Y)" represents the fact that there is a queen in the position (X,Y).

% Domain predicate: the dimension of the board.

```
d(1..n).
q(X,Y) :- assigned(X,Y),d(X;Y).
% Exactly one queen in each row
1 { q(X,Y):d(Y) } 1 :- d(X).
% Exactly one queen in each column
1 { q(X,Y):d(X) } 1 :- d(Y).
% Exactly one queen on each diagonal
q(1+I, R+I) : d(1+I;R+I):e(I) } 1 :- d(R).
{ q(1+I, R-I) : d(1+I;R-I):e(I) } 1 :- d(R).
{ q(C+I, 1+I) : d(C+I;1+I):e(I) } 1 :- d(C).
{ q(C+I, n-I) : d(C+I;n-I):e(I) } 1 :- d(C).
```

A.5 Vertex Cover

The graph is given as facts "vtx(X)." and "edge(X,Y)" and k is the upper bound for vertices in a cover. The main predicate is "incover". Atom "incover(X represents the fact that vertex X is in a vertex cover.

```
%Select at least one of the ends of an edge to be included in the cover
1 { incover(X), incover(Y) } :- edge(X,Y).
% Eliminate models where k+1 or more vertices have been chosen
:- k+1{ incover(X):vtx(X) }.
```

Bibliography

[1] Antonis C. Kakas, Fariba Sadri (Eds.): Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I. *Lecture Notes in Computer Science 2407 Springer 2002*, ISBN 3-540-43959-5

[2] R. E. L. Aldred, S. Bau, D. A. Holton, Brendan D. McKay, Nonhamiltonian 3-Connected Cubic Planar Graphs. *SIAM Journal on Discrete Mathematics Volume 13, Number 1, Page. 25-32.*

[3] H. Andreka and I. Nemeti. The generalized completeness of Horn predicate logic as a programming language. *Acta Cybernetica, 4:3-10,1978.*

[4] Baba laboratory *http://www.geocities.com/babalabo/Hamilton/Ariadne.html*

[5] D. East, M. Truszczynski, DATALOG with constraints - an answer-set programming system *Proceedings of AAAI-00, 2000.*

[6] Debroah. East, DATALOG with Constraints: a new Answer Set Programming Formalism, PhD dissertation. *http://www.cs.engr.uky.edu/etd/theses/uky-cocs-2001-d-001/*

[7] D. East, M. Truszczynski, Propositional satisfiability in answer-set programming, *Proceedings of KI-2001, LNAI 2174, Springer Verlag, 2001.*

[8] M.R. Gary and D. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[9] Gelfond, M., and V. Lifschitz. 1988. The Stable Model Semantics for Logic Programming. *In Proceedings of the 5th International Conference on Logic Programming, 1070–1080. Seattle, USA*, August. The MIT Press.

[10] Ikka Niemelä, Patrik Simons and Tommi Syrjänen. Smodels: A system for Answer Set Programming. *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, April 9-11, 2000, Breckenridge.*

[11] Ilkka Niemelä and Patrik Simons. Extending the Smodels System with Cardinality and Weight Constraints. In *J. Minker, editor, Logic-Based Artificial Intelligence, pages 491–521. Kluwer Academic Publishers, 2000.*

[12] Ilkka Niemelä and Patrik Simons. Extending the Smodels System with Cardinality and Weight Constraints. *Logic-Based Artificial Intelligence, pages 491-521.* Kluwer Academic Publishers, 2000.

[13] I. Niemelä, Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm.*Annals of Mathematics and Artificial Intelligence 25(3,4):241-273, 1999.*

[14] J.M. Crawfard and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *AAAI-93*, 1993

[15] J.M. Crawfard and L.D. Auton. Experimental results on the crossover point in random 3-SAT. In *Artificial Intelligence, 81*, 1993

[16] Lifschitz, V. 1999. Answer set planning. *In Proceedings of ICLP-99, 23–37.*

[17] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing, pp. 365-387, 1991.*

[18] P. Simons, I. Niemelä and Timo Soininen. Extending and Implementing the Stable Model Semantics.*Artificial Intelligence, 138(1-2):181-234, 2002.*

[19] R. Kowalski. *Logic for problem solving.* Elsevier North Holland, New York, NY, 1979.

[20] R.M. Smullyan. *First order logic.* Berlin: Springer-Verlag,1968.

[21] Tommi Syrjänen. Lparse a a procedure for grounding domain-restricted logic programs. *http:// www.tcs.hut.fi/Software/smodels/lparse/, 1999.*

[22] Thomas Eiter, V.S. Subrahmanian, and T.J. Rogers. *Heterogeneous Active Agents, III: Polynomially Implementable Agents*, May 1999.

[23] Tommi Syrjnen. Implementation of local grounding for logic programs with stable model semantics. *Technical Report B18, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland*, October 1998.

[24] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. *Proceedings of the Seventh International Conference pp. 74–84, Morgan-Kaufmann, 2000.*

[25] W. Marek, A. Nerode, and J.B. Remmel. The stable models of predicate logic programs. *Journal of Logic Programming*, 21(3):129-154, 1994.

[26] Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Experimenting with Heuristics for Answer Set Programming. *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI) 2001, pages 635-640, Seattle, WA, USA, August 2001.* Morgan Kaufmann Publishers.

[27] Z. Lonc, M. Truszczynski, Computing stable models: worst-case performance estimates, *Proceedings of ICLP-2002, Peter J. Stuckey, ed., LNCS, Springer-Verlag, Berlin Heidelberg 2002.*

Vita

Date and Place of Birth: Ujjain, India, June 20, 1979

Education: Bachelor of Engineering in Computer Engineering
Rajiv Gandhi Technical University, Indore, 2001

Professional Positions: Graduate Research Assistant
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 2001-2003

Honors: Research Assistantship
University of Kentucky, 2001-2003

Professional Publications:

"The ion transporter superfamily" In Journal *Biochimica et Biophysica Acta 2003* devoted to section Biomembranes.

Soumya Singhi