ABSTRACT OF DISSERTATION

Lengning Liu

The Graduate School

University of Kentucky

2006

COMPUTATIONAL TOOLS FOR SOLVING HARD SEARCH PROBLEMS

ABSTRACT OF DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements of the degree of Doctor of Philosophy in the
College of Arts and Sciences at the University of Kentucky

By

Lengning Liu

Lexington, Kentucky

Director: Dr. Mirosław Truszczyński, Department of Computer Science

Lexington, Kentucky

2006

ABSTRACT OF DISSERTATION

COMPUTATIONAL TOOLS FOR SOLVING HARD SEARCH PROBLEMS

The goal of this dissertation is to develop computational tools for solving hard search problems. We focus on a declarative approach in which we write programs to capture constraints of search problems rather than the step-by-step algorithms that solve the search problems.

Once we write the declarative program that captures a search problem, we want to compute the solutions of the search problem as well. We realize this goal by applying specially designed software called **solvers** on the declarative program we have written. Efficient algorithms and implementations of the solvers make the declarative approach practical in solving hard search problems.

In the dissertation, we investigate one such declarative programming formalism called **logic programming with stable model semantics**. In this formalism, a program is a collection of rules. Rules are built of **monotone** (or **convex**) **abstract constraint atoms**. This formalism extends the normal logic programming with stable model semantics, which has been studied by the community for decades and has rich theories and practical applications. We show in the thesis that the stable model semantics and properties, especially those that are concerned with computation of stable models, extend to logic programs with monotone (or convex) abstract constraint atoms.

***Lparse* programming**, a version of logic programming with stable model semantics, was proposed in 1990's and has been shown to be an effective programming formalism for solving search problems. $Lparse$-programs also extends normal logic programs with specific constraint atoms: **pseudoboolean** constraints. We show in the thesis that pseudoboolean constraints are convex. Thus the theoretical results we obtain for logic programs with convex abstract constraint atoms instantiate to $lparse$-programs. Based on these results, we design a solver that computes stable models of $lparse$-programs via pseudoboolean solvers.

We also study the propositional logic extended with pseudoboolean constraints, a byproduct of our research on $lparse$-programs. We designed and implemented a family local search solvers that compute models of theories in this logic.

We performed experimental study on the solvers we developed. The results show that our solvers are efficient in solving many NP-hard search problems.

KEYWORDS: Knowledge representation, Answer-set programming, Propositional satisfiability, Pseudo-boolean satisfiability, Stochastic local search

COMPUTATIONAL TOOLS FOR SOLVING HARD SEARCH PROBLEMS

By

Lengning Liu

Dr. Mirosław Truszczyński
Director of Dissertation

Dr. Grzegorz W. Wasilkowski
Director of Graduate Studies

19 May 2006

RULES FOR THE USE OF DISSERTATIONS

Unpublished dissertations submitted for the Master's and Doctor's degrees 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 dissertation in whole or in part requires also the consent of the Dean of the Graduate School of the University of Kentucky.

A library which borrows this dissertation for use by its patrons is expected to secure the signature of each user.

Name	Date

DISSERTATION

Lengning Liu

The Graduate School

University of Kentucky

2006

COMPUTATIONAL TOOLS FOR SOLVING HARD SEARCH PROBLEMS

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements of the degree of Doctor of Philosophy in the
College of Arts and Sciences at the University of Kentucky

By

Lengning Liu

Lexington, Kentucky

Director: Dr. Mirosław Truszczyński, Department of Computer Science

Lexington, Kentucky

2006

DEDICATION

To my loving parents,

Qian Situ and Xuhua Liu.

You let me know the meaning of life, family, and love.

ACKNOWLEDGMENTS

First of all, I would like to express eternal gratitude to my adviser, Dr. Mirosław Truszczyński, for his great advice and kind support throughout my graduate study at the University of Kentucky. I could not imagine a better adviser and friend for me than Dr. Truszczyński. He changed me from a student to a researcher. His broad knowledge in math, logic, and computer science, and deep understanding in the research areas are the great help and the most significant factor to the success of my dissertation.

I also would like to thank my committee members: Dr. Raphael Finkel, Dr. Carl Lee, Dr. Victor Marek and Dr. Qiang Ye for reading the drafts of this dissertation and helping me to revise the thesis to a much better shape.

I want to thank Dr. Alexander Dekhtyar, Dr. Raphael Finkel, Dr. Judy Goldsmith and Dr. Victor Marek for deep and fruitful discussions on my research during these years. I always feel lucky to be surrounded by so many talented researchers. I learned a lot from you. Thank you!

Carol Hannahs, Bryan Crawley, Soumya Singhi, Wenzhong Zhao, Emil Iacob, Gayathri Namasivayam, Krol Kevin Mathias, Liangrong Yi, and Peng Dai, thank you for sharing six great years with me. I enjoy the time we spent together, no matter if it is laughing or fighting (I mean, fighting for our opinions during the discussions).

My special thanks go to my dear little sister, Bingyue Liu. Without you, I cannot imagine I would leave our parents and come here to pursue my PhD. Thank you for taking care of our family while I am not there.

Last but not least, I want to thank my lovely wife, Yurong He, for her selfless support and encouragement during these years. Without you, I would not have done this work. Thank you!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The goal of this thesis is to develop computational tools for solving hard search problems represented as $lparse$-programs with weight constraints. Specifically, our goal in this thesis is to develop theories regarding the properties of such programs and investigate effective ways to compute stable models of $lparse$-programs.

1.1 Motivation

Computers were created to help humans to solve problems. The problem may be as simple as evaluating arithmetic expressions or as hard as landing spaceships on moon. In this thesis, we focus on a particular type of problems called **search problems**. In a nutshell, a search problem is defined by a set of conditions (or **constraints**) that solutions to the problem must satisfy. Search problems range from theoretical problems such as graph 3-coloring to practical problems such as scheduling and planning. The difficulty of search problems varies. We consider NP*-hard* search problems in this thesis. Since they are NP-hard, unless $NP = P$, there do not exist polynomial-time bounded algorithms to solve arbitrary instances of these problems. However, this result does not imply that the algorithms will be ineffective on every instance of the problems. In fact, there are algorithms that can solve some instances of NP-hard search problems efficiently.

Since NP-hard search problems appear in many practical applications, modeling and solving them are important and challenging. In this dissertation, we adopt a declarative approach to solve search problems. That is, in order to solve a search problem, we first establish a set of variables, each with its domain of values. These variables represent the "properties" of solutions to the search problem. Then we represent constraints on solutions of the problem as constraints on values of those variables. Next, we compute value assignments to the variables that satisfy all the constraints we just built. Finally, we recover solutions to the original search problem from the valid value assignments we find.

Based on this general approach, researchers have developed several formalisms for solving search problems. We list some of them here:

1. Linear programming, integer programming, and 0-1 integer programming, rooted in operations research. In these formalisms, we represent constraints as linear equalities or inequalities. Depending on whether linear, integer, or 0-1 integer programming is used, variables have real, integer, or $\{0, 1\}$ domains.

2. Constraint satisfaction

3. Propositional satisfiability (or SAT)

4. Logic programming with stable-model semantics, rooted in knowledge representation and non-monotonic reasoning

We focus on logic-based formalisms in this dissertation. In particular, we study the formalism of logic programming with stable-model semantics.

Gelfond and Lifschitz define stable-model semantics of normal logic programs, a subclass of logic programs that have a simple form (we will introduce it in the next chapter) [59]. Because of its ability to deal with incomplete knowledge through default negation, normal logic programming with stable-model semantics has become an effective knowledge representation tool. Marek, Truszczyński [101], and Niemelä [109] propose a programming paradigm based on normal logic programming with stable-model semantics. In this programming paradigm, we represent search problems as logic programs and solve them by computing the stable models of the logic programs. Simons *et al.* [125] extend normal logic programming with explicit constructs to model numerical constraints, called **weight constraints**, which occur commonly in search problems. With the help of weight constraints, we can represent many search problems more concisely in this extended formalism, compared to their normal logic-programming representation.

To realize this programming paradigm, researchers have developed software including programming front-ends and computational back-ends. A typical programming front-end

uses a high-level language (usually a fragment of first-order logic) that contains predicates, function symbols, variables and their domains. The language often provides direct constructs called **weight constraints** to facilitate modeling numerical constraints. Using this language, we write the high-level logic program to capture the constraints of a search problem. The constraints defining a search problem are generic to the search problem and independent of the specific data instance. One of the benefits of writing logic programs in a high-level language is that we can *separate our programs from data* [1]. That is, the program, with the help of predicates and domain variables, models the generic constraints of a search problem. Then the data that correspond to an instance of the search problem are given as a set of facts represented as predicates over the domain values of variables. We **ground** the program and the data using a specially designed program called the **grounder**. In a nutshell, the grounder takes the high-level encoding of the problem and replaces the variables in the encoding by their possible values defined in the input data. After this step, we get a ground logic program. Then we can use a **solver** to compute stable models of the ground logic program. The solver is the actual back-end computing machinery that solves the search problem. Finally we can recover the solutions to the original search problem from the stable models.

To summarize, we take the following steps to solve a search problem using this formalism [34, 101, 109]:

1. modeling: We write a logic program Π_s in a high-level logic programming language that captures the generic constraints of the target search problem s. We also construct the data set D_d that corresponds to an instance d of problem s.

2. grounding: We execute a grounder program with input (Π_s, D_d). The grounder generates a ground program $P_{s,d}$, which is a propositional version of the combination of logic program Π_s and the data D_d. We are guaranteed that stable models of $P_{s,d}$ correspond to solutions to the instance d of the search problem s. The implementation

[1] This separation may not be complete for the encodings of certain search problems. Nevertheless, using the high-level language, we can still exploit the concept of a program and the input data to some degree in those cases.

of the grounder is independent of the search problem s or the instance d.

3. solving: We call a solver program with $P_{s,d}$ as input. The solver computes the stable models of $P_{s,d}$, from which we can recover solutions to the search problem.

The whole procedure to represent and solve a search problem is shown in Figure 1.1. In this thesis, we focus on the shaded area in the picture.

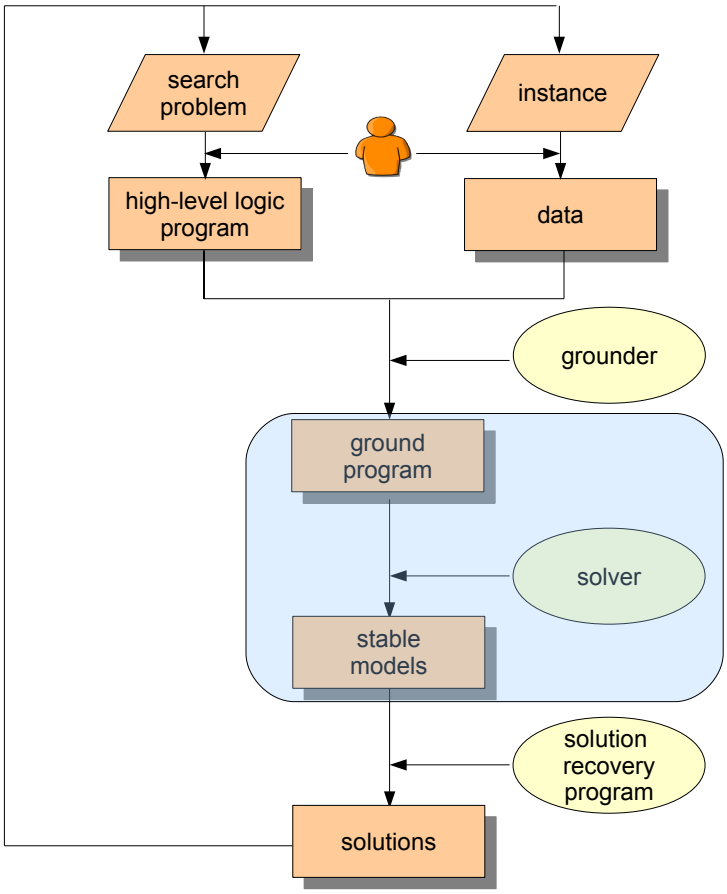

Figure 1.1: Logic programming paradigm

One implementation of this logic programming paradigm is **SMODELS** [114]. The high-level language in **SMODELS** allows the use of weight constraints. We denote the ground programs in **SMODELS** by $lparse$-programs. Simons *et al.* [125] have developed a solver called *smodels* for $lparse$-programs.

Lparse-programs have received much attention in the logic programming community. Several other solvers have emerged in recent years, including *cmodels* [7] and *assat* [81].

Another implementation of logic programming with stable-model semantics is **DLV** [20, 76]. The high-level language used in **DLV** provides direct constructs, called **aggregates**, for modeling numerical constraints. In particular, weight constraints are a class of aggregates allowed in **DLV**. Ground programs in **DLV** are **disjunctive logic programs**, which are more general than *lparse*-programs. In this thesis, we focus on *lparse*-programs only [2] and do not discuss disjunctive logic programs.

We now present an example to show how to represent a search problem as an *lparse*-program. At this point, the syntax is not important. The idea is to illustrate the declarative approach we have been discussing so far.

Definition 1. *Let $G = (V, E)$ be an undirected graph. A set $D \subseteq V$ of vertices of G is* **dominating** *if for every vertex $x \in V$, either $x \in D$ or there exists at least one neighbor y of x such that $y \in D$. We call such a subset D a* **dominating set** *of G.*

Problem 1. Given a graph $G = (V, E)$ and an integer k, find a dominating set of G of size at most k. (The decision problem implied by this problem is NP-complete [58]). △

***Lparse*-program.** We use the following set of atoms in this encoding: in_x for $x = 1, \ldots, n$, where $n = |V|$. The intended meaning of in_x is that vertex x is in the dominating set D. The following *lparse*-program encodes Problem 1:

1. $\{in_1, \ldots, in_n\}k$

   This rule ensures that the size of D is at most k.

2. $\leftarrow \mathbf{not}(in_x), \mathbf{not}(in_{y_1}), \ldots, \mathbf{not}(in_{y_{m_x}})$

   for every $1 \leq x \leq n$ and for every neighbor y_j of x (there are m_x neighbors)

   These rules ensure the defining constraint for a dominating set. Symbol **not** denotes the default negation, which we introduce in Chapter 2.

[2] We actually assume a simplified version of *lparse*-programs where all weights in the weight constraints are non-negative. We will discuss this matter in more detail later in the thesis.

We stress again that programmers do not write their $lparse$-programs directly. All $lparse$-programs are constructed by the grounder from their high-level representations.

The elegant semantics and the mature logic programming front-end make **SMODELS** a practical programming environment. Therefore it is important to study properties of $lparse$-programs. It is also important and challenging to develop efficient solvers that compute stable models of $lparse$-programs because computing stable models is the key to apply this approach in practical applications — we not only want to represent a problem, but also want to *solve* the problem.

1.2 Goals of this thesis

To this end, this thesis considers a setting that is more general than $lparse$-programs: logic programs that are built of abstract constraint atoms. We assume these abstract constraint atoms satisfy the **monotonicity** or the **convexity** property. Under this setting, our first sub-goal is to extend properties that have been proved for normal logic programs to this abstract case. Since the weight constraints in $lparse$-programs are convex [3], all the abstract properties are valid in $lparse$-programs as well.

Our second sub-goal is to use the properties of $lparse$-programs to develop a new method to compute stable models of $lparse$-programs. We want to compute stable models of $lparse$-programs better than *smodels*.

1.3 Related work

Our work is related to propositional satisfiability (or SAT) and pseudoboolean satisfiability (or PB SAT).

Propositional satisfiability tests whether a propositional logic formula in the conjunctive normal form (CNF) is satisfiable. This decision problem is a well-known NP-complete problem when each disjunction in the formula contains at least three literals. One research direction in the SAT community is to develop efficient algorithms for testing satisfiability of a propositional logic formula.

[3] with the assumption that all weights in the weight constraints are non-negative

Davis *et al.* [19] first proposed an algorithm (DPLL) based on resolution to compute models of a formula in CNF. The DPLL algorithm explores an enormous search space. A naive implementation often has difficulty in solving a CNF formula with as few as 50 propositional atoms. Therefore, early implementations of DPLL algorithm were not successful in propositional satisfiability testing.

A performance break-through occurred in the 1990's. From 1990's to the present, researchers have devoted much effort to the development of fast implementations of the DPLL algorithm. Researchers propose several techniques to improve the performance of the DPLL algorithm. They includes good heuristics for atom selection [77, 105, 134], clause learning [105], non-chronological back-tracking [105], and watched literals for boolean constraint propagation [134]. These techniques, combined with the increased computational power of computers, yield fast implementations that can solve instances having hundreds or even thousands of atoms and tens of thousands or hundreds of thousands of clauses.

In 1990's, Selman *et al.* [123] proposed a completely different type of algorithms that also compute models of propositional formulas in CNF. This type of algorithm is known as **stochastic local search** (or SLS) algorithms. Unlike DPLL-based algorithms, SLS algorithms are **incomplete**, meaning that they may not be able to find a model of an input theory even if there is one. However, their ability to compute models of large satisfiable theories, which are often beyond the power of DPLL based solvers, makes them attractive. Current implementations of SLS algorithms are often capable of solving instances that have hundreds of thousands of variables and millions of clauses.

With these developments, **SAT solvers** become applicable in many practical applications such as scheduling and planning, hardware or protocol verification.

A drawback of SAT solvers is that they require an input theory to be in CNF. Constraints defining search problems of practical importance often do not have a direct and compact representation as formulas in CNF and in many cases require large sets of clauses to be faithfully described. Constraints involving numeric values, typically modeled as linear inequalities, are such constraints. The large size of CNF theories representing search

problems limits the effectiveness of SAT solvers.

To circumvent the size explosion problem in representing numerical constraints as clauses, researchers have studied constraints that are more general than propositional clauses and are attuned to constraints commonly appearing in applications. Certain integer programming constraints, called **pseudoboolean** (or PB) constraints, have received particular attention [11, 12, 28, 67]. This research results in several solvers of pseudo-boolean constraints [121, 2, 96, 132].

With the great success of SAT solvers, researchers in the logic programming community have attempted to use SAT solvers to compute stable models of logic programs [18, 7, 50, 81]. Specifically, they establish theoretical results that transform a normal logic program into a SAT instance (a propositional logic formula in CNF) so that models of the SAT instance are precisely the stable models of the original logic program. This work makes it possible to use off-the-shelf SAT solvers to compute stable models of normal logic programs. $Lparse$-programs extend normal logic programs with weight constraints. Ferraris and Lifschitz [52] establish a connection between $lparse$-programs and SAT instances by compiling away weight constraints in an $lparse$-program and obtain a normal logic program. The compilation involves a set of additional normal-logic program rules and a set of auxiliary atoms, helping to avoid a size explosion. Nevertheless, the size of the resulting normal-logic program is still larger than the original $lparse$-program. Furthermore, the additional logic program rules and atoms complicate the logic program. Both factors have a negative influence on the underlying SAT solvers in terms of the amount of time needed by those solvers to find satisfying truth assignments.

1.4 Contributions of the thesis

To achieve the goal of building effective computational tools to solve search problems represented as logic programs, we follow the general idea implemented in *cmodels* [7] and *assat* [81]. Both *cmodels* and *assat* convert logic programs into propositional logic theories and use SAT solvers to compute models of the propositional logic theories. The way by which they convert logic programs into propositional logic theories guarantees that models

the SAT solvers compute are precisely the stable models of the original logic programs. As we have mentioned, *assat* computes stable models only for normal logic programs. *cmodels* accepts $lparse$-programs as input but has to compile away weight constraints in $lparse$-programs, which is not efficient.

One of the major contributions of this thesis is **a new method** to compute stable models of arbitrary $lparse$-programs. The key difference between our work and *cmodels* and *assat* is that we use PB SAT solvers instead of SAT solvers to compute stable models of $lparse$-programs.

This work is motivated by the active development of PB SAT solvers and the fact that the PB constraints used in PB SAT instances and the weight constraints used in $lparse$-programs are similar. We show **a direct transformation** from $lparse$-programs to PB SAT instances such that stable models of $lparse$-programs are precisely models of PB SAT instances. To establish this correspondence, we first need to establish properties of $lparse$-programs. In particular, we **extend the theoretical results** [81] that help converting a normal logic program into a SAT instance. We **propose an extension** to propositional logic in which PB constraints can appear in a clause. We call this logic PL^{wa}. We use this logic as the counterpart of propositional logic used in *cmodels* [7] and *assat* [81]. Finally, we convert the resulting theory in logic PL^{wa} into PB SAT instances and apply PB SAT solvers there.

The contributions of this thesis include:

1. We establish properties for logic programs built of abstract constraint atoms [102]. This is a generalization of $lparse$-programs when the abstract constraints are **monotone**. We refer to logic programs built of monotone abstract constraint atoms mac-programs. We develop a collection of theories regarding the properties of mac-programs. They include:

   (a) strong and uniform equivalence of mac-programs

   (b) Fages' Lemma for mac-programs

   (c) completion of mac-programs (for this purpose, we propose an extension to

propositional logic where formulas are boolean combinations of monotone abstract constraint atoms)

(d) loop formulas of mac-programs

We also propose a syntactical variant of mac-programs where abstract constraints are **convex** instead of monotone. The purpose of proposing this variant is that convex constraints align better with $lparse$-programs than mac-programs. We show that all the properties we have proved for mac-programs hold for logic programs built of convex constraint atoms as well.

2. **Weight constraints** are **convex**. Therefore, all properties we proved for logic programs built of convex constraint atoms project to $lparse$-programs. Based on these properties, especially the properties concerned with Fages' Lemma, completion, and loop formulas, we design and implement a new method, denoted by $pbmodels$, to compute stable models of $lparse$-programs. This new method uses PB SAT or PL^{wa} SAT solvers to compute stable models. It differs from $cmodels$ [7] because it does not compile away weight constraints from $lparse$-programs.

3. We design and develop the first stochastic local search (SLS for short) algorithms for arbitrary PL^{wa}-theories. Our algorithms, called $wsat(wa)$, follow the existing work in the literature [68, 72, 123]. The development of this solver is motivated by the fact that the completion and loop formulas of $lparse$-programs are logic PL^{wa}-theories. We can use $wsat(wa)$ as a back-end solver for $pbmodels$. However, $wsat(wa)$ is of interest itself due to the importance of PB-constraints in problem modeling.

4. We perform an extensive experimental study on implementations of the algorithms we propose in the thesis. We test our solvers, including the new $lparse$-program solver — $pbmodels$, and a family of SLS solvers — $wsat(wa)$ for PL^{wa}-theories. We compare our implementations to existing solvers in the literature. The experimental study completes and validates the thesis by showing our algorithms perform better than other solvers in solving a number of NP-hard search problems.

1.5 Organization of the thesis

The thesis is organized as follows: Chapter 2 introduces preliminary definitions for our latter discussion; Chapter 3 studies properties of $lparse$-programs and proposes an extension to propositional logic called logic PL^{wa}. Some of these properties establish the connection between $lparse$-programs and logic PL^{wa} theories. Based on the theoretical results in this chapter, we develop a new solver for $lparse$-programs with the help of PB SAT solvers; Chapter 4 proposes a family of SLS algorithms for logic PL^{wa}; Chapter 5 shows our experimental results of comparing our work to existing work in the field on several search problems; finally, Chapter 6 concludes our work and discusses possible future research directions.

Chapter 2

Logic programming with stable-model semantics

In this chapter, we introduce basic terminology used throughout this thesis. In particular, we define logic program and the stable-model semantics of logic programs. These notions are the basis for this thesis. This chapter is organized as follows: Section 2.1 introduces normal logic programming and Section 2.2 introduces a widely accepted extension to normal logic programming that represents certain numerical constraints directly. All the material presented in this chapter was developed by the research community between 1980 and 2002.

2.1 Normal logic programming with stable-model semantics

Logic programming with stable-model semantics is a declarative programming formalism for knowledge representation and for solving search problems.

Definition 2. *A* **normal logic program rule** *is an expression r of the form:*

$$a \leftarrow b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n). \tag{2.1}$$

where a, b_i and c_j are all propositional atoms. A **normal logic program** *is a collection of rules of the form (2.1).*

We call a the **head** of r, and the set $\{b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n)\}$ the **body** of r[1].

Let I be a set of atoms. The set I can be regarded as a representation of the following truth assignment:

1. atom a gets value **true** (or I satisfies a), written $I \models a$, if $a \in I$;

2. atom a gets value **false** (or I does not satisfy a), written $I \not\models a$, if $a \notin I$.

[1] Sometimes we view the body of a rule as the **conjunction** of its literals.

We say that I is a **model of** (or **satisfies**) a rule r of the form (2.1), written $I \models r$, if $a \in I$ whenever $b_i \in I$ $(i = 1, \ldots, m)$ and $c_j \notin I$ $(j = 1, \ldots, n)$. Next, I is a **model of** (or **satisfies**) a program P, denoted by $I \models P$, if I is a model of every rule in P.

In the setting of propositional logic, a propositional logic theory can be viewed as a concise representation of *all* of its models. For example, a theory consisting of the single clause $\{a \vee b \vee c\}$ represents seven models that satisfy this clause. In this case, we say the meaning, or more formally the **semantics**, of a propositional logic theory is given by the set of all its models.

In normal logic programming, we often view a normal logic program as a representation of a subset of its models. Different semantics may adopt different subsets of models. **Stable-model semantics** is one of the mostly studied and widely accepted semantics for normal logic programs.

In stable-model semantics, symbols $\leftarrow$ and $\mathbf{not}$ are not the same as the material implication $\rightarrow$ and the logical negation $\neg$ in propositional logic. In particular, $\mathbf{not}$ is called **negation as failure**. Intuitively, $\mathbf{not}(a)$ is true if we cannot derive (or prove) a is true.

We first introduce a special class of normal logic programs. A normal logic program rule (2.1) is **Horn** if $n = 0$. A normal logic program is **Horn** if every rule in it is Horn. Horn normal logic programs have the following properties.

Let P be a normal logic program and M a set of atoms. We write $P(M)$ to denote the set of rules in P whose bodies are satisfied by M. Then we have the following proposition.

Proposition 1. *Let P be a Horn normal logic program and M_1, M_2 two sets of atoms. If $M_1 \subseteq M_2$, then $P(M_1) \subseteq P(M_2)$.*

Proof. Let r be an arbitrary rule in $P(M_1)$. Therefore, M_1 satisfies the body of r. Since $M_1 \subseteq M_2$ and the body of r does not have $\mathbf{not}$, M_2 satisfies the body of r as well. That is, $r \in P(M_2)$. □

We define a **bottom-up computation** $\langle X_n \rangle_{n=0}^{\infty}$ of a Horn normal logic program P as follows:

1. Let $X_0 = \emptyset$;

2. $X_{n+1} = \{a\colon$ there exists $r \in P(X_n)$ such that a is the head of $r\}$

It is clear that the bottom-up computation of a Horn normal logic program is unique.

The bottom-up computation has the following property.

Proposition 2. *Let P be a Horn normal logic program and $\langle X_n \rangle_{n=0}^{\infty}$ its bottom-up computation. Then $M = \bigcup_{n=0}^{\infty} X_n$ is a model of P.*

Proof. Let r be an arbitrary rule in P. If M does not satisfy the body of r, then M satisfies r. Therefore, we assume M satisfies the body of r. Since

$$M = \bigcup_{n=0}^{\infty} X_n$$

there exists an X_n such that X_n satisfies the body of r. Then by the definition of the bottom-up computation, X_{n+1} contains the head of r. Therefore $M \models r$ as well. It follows that M is a model of P. □

Theorem 1. *Every Horn normal logic program P has a least model.*

Proof. We show that the union of all X_n's in the bottom-up computation of P is the least model. Let $\langle X_n \rangle_{n=0}^{\infty}$ be the bottom-up computation of P and $M = \bigcup_{n=0}^{\infty} X_n$. By Proposition 2, M is a model of P. To show it is the least model of P, it is sufficient to show that, for an arbitrary model M' of P, $M \subseteq M'$.

Assume it is not the case. That is, there exists a model M' of P such that $M \not\subseteq M'$. Let $X = M \setminus M'$. Since $M = \bigcup_{n=0}^{\infty} X_n$, for every $a \in X$, there exists X_n for some $n = 0, 1, \ldots, infty$ such that $a \in X_n$. Let a_0 be the atoms in X such that its corresponding X_m has the smallest index in the sequence $\langle X_n \rangle_{n=0}^{\infty}$.

That is, for every $i = 0, 1, \ldots, m-1$, $X_i \subseteq M$ and $X_i \subseteq M'$. Furthermore, $X_m \subseteq M$ and $X_m \not\subseteq M'$.

Therefore, there exists a rule $r \in P$ such that $r \in P(X_{m-1})$ and a is the head of r. Since $X_{m-1} \subseteq M'$ and P is Horn, $r \in P(M')$ as well. Since M' is a model of P, $M' \models a$. It contradicts the assumption that $a \notin M'$. Therefore, the theorem follows. □

We denote this least model of P by $lm(P)$.

Next, we consider the case of arbitrary normal logic programs. Gelfond and Lifschitz [59] introduced the notion of a **reduct**, which is the key to the definition of a stable model of an arbitrary normal logic program.

Definition 3. *Let P be a normal logic program and M a set of atoms. The* **reduct** *of P with respect to M, denoted by P^M, is a normal logic program obtained by:*

1. *removing from P all rules (2.1) such that $c_i \in M$ for some $i = 1, \ldots, n$; and*
2. *removing from the remaining rules all $\mathbf{not}(c_i)$'s, $i = 1, \ldots, n$.*

Clearly the reduct of any normal logic program P with respect to M is Horn. Therefore there exists a least model $lm(P^M)$ of the reduct.

Definition 4. *Let P be a normal logic program. The set M of atoms is a* **stable model** *of P if $M = lm(P^M)$.*

The following theorem shows that the definition of stable models is valid.

Theorem 2. *Let P be a normal logic program. Then stable models of P are models of P.*

Proof. Let M be a stable model of P. Then $M = lm(P^M)$. Let r be an arbitrary rule of the form 2.1 in P. There are two cases:

1. The body of r contains some c_i such that $c_i \in M$. Then it is clear $M \models r$.
2. Every c_i in the body of r does not belong to M. Then $r' = a \leftarrow b_1, \ldots, b_m$ (the rule got from r by removing all $\mathbf{not}(c_i)$'s) belongs to P^M. Since $M = lm(P^M)$, $M \models P^M$. Since $r' \in P^M$, $M \models r'$. It follows that $M \models r$.

Since r is arbitrary, $M \models P$. □

We note, however, the converse implication of this theorem does not hold. Here is an example:

Example 3. *Let P be the normal logic program consisting of the following rule:*

$$p \leftarrow p.$$

Clearly both $\emptyset$ and $\{p\}$ are models of P. However, the only stable model of P is $\emptyset$. We can verify that the reduct $P^{\{p\}} = P$. The least model of the reduct is $\emptyset \neq \{p\}$. $\triangle$

For Horn programs, the least model coincides with the unique stable model.

Theorem 4. *Let P be a Horn normal logic program. Then $lm(P)$ is the only stable model of P.*

Proof. If P is a Horn normal logic program, then $P = P^M$ for any M since the rules in P do not contain **not**. Then M is a stable model of P if and only $M = lm(P^M) = lm(P)$. Since the least model is unique, it is the only stable model of P. $\square$

We now give an example of stable models of a normal logic program.

Example 5. *Let $P = \{a \leftarrow \mathbf{not}(b). \quad b \leftarrow \mathbf{not}(a).\}$. Let $M_1 = \{a\}$. We can verify that $P^{M_1} = \{a \leftarrow .\}$. Therefore, M_1 is a stable model of P since the least model of P^{M_1} is M_1. We also observe that M_1 is a model of P. On the other hand, $M_2 = \{a, b\}$ is not a stable model of P (note that $P^{M_2} = \{\}$), even though it is a model of P.* $\triangle$

Finally we note that, if a normal logic program P contains a rule r of the form:

$$f \leftarrow b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n), \mathbf{not}(f) \tag{2.2}$$

where f is an atom that does not appear anywhere else in P, then if M is a stable model of P, M cannot satisfy $b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n)$. Assume it is not the case. That is M is a stable model of P such that $b_i \in M$ for $i = 1, \ldots, m$, and $c_j \notin M$ for $j = 1, \ldots, n$. If $f \in M$, then $f \leftarrow b_1, \ldots, b_m \notin P^M$. Then f cannot belong to the least model of the reduct since f does not appear anywhere else in P. This result contradicts the fact that $M = lm(P^M)$. If $f \notin M$, then since $c_j \notin M$ for $j = 1, \ldots, n$, $f \leftarrow b_1, \ldots, b_m \in P^M$. Since $M = lm(P^M)$ and $M \models b_1, \ldots, b_m$, $f \in lm(P^M)$. However, $f \notin M$. It is also a contradiction. Therefore, no stable model of P satisfies $b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n)$.

We write rules of the form (2.2) as

$$\leftarrow b_1, \ldots, b_m, \mathbf{not}(c_1), \ldots, \mathbf{not}(c_n)$$

and call them **constraint rules**.

Example 6. *Let us continue with Example 5. Let*

$$P' = P \cup \{\leftarrow a\}.$$

Then P' has only one stable model now: $\{b\}$. Indeed, P has two stable models $\{a\}$ and $\{b\}$. Since $\{a\}$ satisfies the body of the constraint rule in P', $\{a\}$ cannot be a stable model of P'. The other stable model of P does not satisfy the body of the constraint rule in P'. Therefore, it is the only stable model of P'. △

2.2 Stable logic programming extended with weight atoms (*lparse*-programs)

Normal logic programming allows only propositional atoms in rules. To capture constraints in problems that involve numerical values, several normal logic program rules are often needed, and constructing them is not straightforward. To facilitate modeling and, later, solving problems that involve such constraints, Simons *et al.* [125] introduced and studied an extension of normal logic programming with weight atoms. We call logic programs in the extended syntax *lparse*-programs [2].

Definition 5. *A* **weight atom** *(w-atom, for short) is an expression W of the form lXu, where X is a set of weighted propositional atoms of the form $[a_1 = w_1, \ldots, a_k = w_k]$, and l, u and w_i's are non-negative integers.*

We call the set $\{a_1, \ldots, a_k\}$ of atoms the **domain** of the w-atom, denoted by $Dom(W)$. Integer w_i is the **weight** of the atom a_i in this weight atom. We call l and u the **lower bound** and the **upper bound** of W respectively. Intuitively, a w-atom represents the constraint that the sum of w_i's (or **weights**) where a_i's are true should be between l and u. Bounds l or

[2] The name comes from the *lparse* grounder program in **SMODELS** [125].

u may be missing, which means we do not constrain the sum of weights from below or above, respectively.

A w-atom lXu is called a **cardinality atom** (c-atom, for short) if all w_i's in X are 1. We write a c-atom as $l[a_1, \ldots, a_k]u$ by omitting all the weights [3]. We observe that we can represent a propositional atom a by the c-atom $1[a]$ and a proposition literal $\neg a$ by the c-atom $[a]0$, so weight atoms generalize cardinality atoms, which in turn generalize propositional literals.

A truth assignment I satisfies a w-atom $l[a_1 = w_1, \ldots, a_k = w_k]u$ if

$$l \leq \sum\{w_i \colon I \models a_i\} \leq u.$$

We call a w-atom **tautological** if it is satisfied by every truth assignment. In particular, if both bounds of a w-atom are missing, then it is tautological. We call a w-atom **contradictory** if no truth assignment satisfies it.

Next, we give the definitions of $lparse$-rules and $lparse$-programs.

Definition 6. *An **lparse-rule** (called a* **weight constraint rule** *in Simons* et al. *[125]) r is an expression of the following form:*

$$A \leftarrow A_1, \ldots, A_n \tag{2.3}$$

where A, $A_1, \ldots, A_n$ are w-atoms.

The **negation as failure** operator $\mathbf{not}$ in normal logic programming is hidden in the upper bound constraints of w-atoms. For example, the c-atom $[a]0$ represents $\mathbf{not}(a)$. We will discuss this matter in more detail in Chapter 3.

We call A the **head** of r, denoted by $hd(r)$, and the set $\{A_1, \ldots, A_n\}$ the **body** of r, denoted by $bd(r)$. We use $hset(r)$ to denote the set of atoms that occur in the head of r. An ***lparse*-program** P is a collection of rules of form (2.3). We denote by $At(P)$ the set of propositional atoms in program P. In the rule, commas "," can be viewed as conjunctions. Similar to the normal logic programming case, an interpretation I **satisfies** a rule of form

[3] For uniformity, we do not use the syntax specified in **SMODELS**, where c-atoms are written as $l\{a_1, \ldots, a_k\}u$.

(2.3) if it satisfies A whenever it satisfies $A_1, \ldots, A_n$. If the head A is missing, the rule becomes a **constraint rule**, which is satisfied by I if and only if I does not satisfy some A_i, $1 \leq i \leq n$, in the body.

We give an example of an $lparse$-program:

Example 7.

$$P = \{2[a = 2, b = 3]3 \leftarrow [a, b]2\}$$

The program P contains one rule. The head of the rule is a w-atom $2[a = 2, b = 3]3$. The body of the rule is a c-atom $[a, b]2$. △

We now define the stable models of $lparse$-programs. The following definitions come from Simons *et al.* [125].

Definition 7. *A rule of the form (2.3) is a* **definite Horn rule** *if the domain of A has the form $1[a]$ and all A_i's in the body do not have upper bounds. A* **definite Horn program** *is a logic program in which every rule is a definite Horn rule.*

In a definite Horn program P, there is always a unique smallest model, denoted by $lm(P)$ [125]. The existence of the unique smallest model is implied by the fact that rules in a definite Horn constraint program are **monotone**. By monotone we mean that if the body of a rule is satisfied by I, then it is also satisfied by any superset of I. Therefore, we can define a consistent bottom-up computation of a definite Horn program P as we did for a Horn normal logic program. Then the union of all X_n's in this bottom-up computation forms the unique smallest model of P as well.

Now let us take an arbitrary $lparse$-program P. We first define the **reduct** of P with respect to a set of atoms M, denoted by P^M.

Definition 8. *Let P be an $lparse$-program and M a set of atoms. The* **reduct** *of P with respect to M, P^M, is defined by*

1. *removing from P all constraint rules;*
2. *removing from P all rules whose bodies contain a w-atom A_i such that the upper bound constraint of A_i is not satisfied by M;*

3. *for each remaining rule r whose head is lXu (l or u may be missing), replacing it with the following set of rules:* $1[p] \leftarrow bd(r)$, *where* $p \in hset(r) \cap M$

4. *for each rule r we get after the previous step, for every w-atom lXu in the body of r, replacing it with a new w-atom lX', where* $X' = [a_i = w_i : a_i = w_i \in X$ *and* $M \models a_i]$.

Example 8. *We continue with Example 7. Let*

$$P = \{2[a = 2, b = 3]3 \leftarrow [a, b]2\}$$

Let $M = \{a\}$. The program P contains one rule. The rule is not a constraint rule. Moreover, the upper bound constraint of the body w-atom is not violated by M. Therefore, the reduct of P w.r.t. M is the following definite Horn program:

$$P^M = \{1[a] \leftarrow [a]\}.$$

Let $M' = \{a, b\}$. Then

$$P^{M'} = \{1[a] \leftarrow [a, b] \quad 1[b] \leftarrow [a, b]\}$$

△

It is clear that the reduct of an $lparse$-program with respect to a set of atoms is always a definite Horn program. Therefore, there exists a unique smallest model in the reduct. If this model coincides with the original set of atoms, and the original set of atoms is a model of the program, then it is called a stable model of the program. Formally we have the following definition.

Definition 9. *Let P be an $lparse$-program and M a set of atoms. Then M is a stable model of P if 1) $M \models P$; and 2) $M = lm(P^M)$.*

Example 9. *Let us take the $lparse$-program P in Example 7 and $M = \{a\}$. The reduct P^M is given in Example 8. Since the c-atom $[a]$ in the body is satisfied by the empty set $\emptyset$, the smallest model of the reduct is $\{a\}$. Therefore, M is a stable model of P. Similarly we can verify that $\{b\}$ and $\{a, b\}$ are both stable models of P.*

If we change the c-atom $[a, b]2$ *in* P *to* $[a, b]1$*, then* $\{a, b\}$ *is no longer a stable model of the resulting program* P'*. The reason is* $\{a, b\}$ *does not satisfy the upper bound constraint of the c-atom* $[a, b]1$*. Therefore, the reduct becomes an empty program, in which the smallest model is the empty set* $\emptyset$*. Since* $\{a, b\} \neq \emptyset$*,* $\{a, b\}$ *is not a stable model of* P'*.* $\triangle$

In Chapter 3, we present a different but equivalent definition for stable models of an $lparse$-program.

Finally, we note that the syntax of programs, in particular the syntax of weight atoms, introduced by Simons *et al.* [125] is more general. It allows both atoms and negated atoms to appear within weight atoms, as well as negative weights (we call such weight atoms **sw-atoms**). As Simons *et al.* [125] showed, negated atoms and negative weights are closely related. In particular, one can use negated atoms to represent negative weights in a w-atom. Then, by introducing new propositional variables, one can also remove occurrences of negated atoms while preserving stable models (modulo newly introduced variables, of course).

Chapter 3

Lparse-programs, stable models, and their properties

In this chapter, we study properties of $lparse$-programs. We pursue it in a more general setting of programs with abstract constraints. In this setting, abstract constraints play the role of w-atoms in $lparse$-programs. To be precise, we focus on a special class of constraints called **monotone** abstract constraints. We refer to these logic programs as mac-programs. This class of logic programs was proposed by Marek *et al.* [97, 102]. We also introduce a related class of programs with **convex** abstract constraints. Both formalisms allow constraints to appear in the heads of program rules, which sets them apart from other recent proposals for integrating constraints into logic programs [26, 21, 49, 117, 118] and makes them suitable as an abstract basis for formalisms such as $lparse$-programs.

Under this abstract setting, we generalize several results from normal logic programming to programs with monotone constraints. We also discuss how these techniques and results can be extended further to the setting of programs with convex constraints.

We show that the notions of uniform and strong equivalence of programs [47, 80, 79, 129] extend to programs with monotone constraints, and that their characterizations [47, 129] generalize, too.

We adapt the notion to programs with monotone constraints of a **tight** program [48] and generalize Fages Lemma [50].

We introduce extensions of propositional logic with monotone constraints. We define the completion of a monotone-constraint program with respect to this logic, and generalize the notion of a loop formula. We then prove the loop-formula characterization of stable models of programs with monotone constraints, extending to the setting of monotone-constraint programs results obtained for normal logic programs by Clark [18] and Lin and Zhao [81].

Programs with monotone constraints make explicit references to the default negation operator. We show that a more general class of constraints, called **convex**, can eliminate

default negation from the language. We argue that all results extend to programs with convex constraints.

Finally we show that programs with monotone and convex constraints have a rich theory that closely follows that of normal logic programming. It implies that programs with monotone and convex constraints form an abstract generalization of extensions of normal logic programs. In particular, all results we obtain in the abstract setting of programs with monotone and convex constraints specialize to $lparse$-programs and, in most cases, yield results that are new.

These results have practical implications. The properties of the program completion and loop formulas, when specialized to the class of $lparse$-programs, yield a method to compute stable models of $lparse$-programs by means of pseudoboolean satisfiability (or PB SAT) solvers [1, 45, 82, 96, 132]. This method follows the path explored by *cmodels* [7] and *assat* [81]. The difference between those two methods and our approach is that those two methods rely on SAT solvers to compute stable models. *Assat* only accepts normal logic programs as the input, so no w-atoms occur in programs. *Cmodels* accepts, theoretically, the full version of $lparse$-programs. However, in practice, it only works when all w-atoms in input programs are c-atoms. Moreover, since *cmodels* relies on compiling away w-atoms in the $lparse$-program (that is, converting an $lparse$-program into an equivalent normal logic program), the extra overhead caused by the compilation greatly affects the effectiveness of the underlying SAT solvers in some cases.

This chapter is organized as follows: in Section 3.1, we introduce mac-programs, the abstraction of $lparse$-programs, and basic concepts such as stable models of mac-programs; in Section 3.2 we prove the equivalence results for mac-programs; in Section 3.3 we extend Fages' Lemma to mac programs. Then we introduce completion and loop formulas of mac programs; in Section 3.4 we introduce a syntactical variant of monotone abstract constraints called **convex constraints**. Convex constraints align better to w-atoms and do not use explicit default negations. We show that all results we proved for mac-programs are also valid in **convex constraint programs**; finally in Section 3.5, we describe the algorithm of $pbmodels$, a new method that computes stable models of$lparse$-programs via PB SAT

solvers.

Our work in this chapter has been published in [86, 85].

3.1 *Mac* programs — a generalization of logic programs with weight constraints

The definitions and results we present in this section come from the work by Marek and Truszczyński [102]. Some of them here are more general than their work because we allow constraints with infinite domains and programs with inconsistent constraints in the heads.

Definition 10. *A* **constraint** *is an expression $A = (X, C)$, where $X \subseteq At$ and $C \subseteq \mathcal{P}(X)$ ($\mathcal{P}(X)$ denotes the powerset of X).*

We call the set X the **domain** of the constraint $A = (X, C)$ and denote it by $Dom(A)$. Informally speaking, a constraint (X, C) describes a property of subsets of its domain, with C consisting precisely of these subsets of X that **satisfy** the constraint (have property) C.

As we have mentioned in Chapter 2, we identify truth assignments (interpretations) with the sets of atoms they assign the truth value **true**. That is, given an interpretation $M \subseteq At$, we have $M \models a$ if and only if $a \in M$. We say that an interpretation $M \subseteq At$ **satisfies** a constraint $A = (X, C)$ ($M \models A$), if $M \cap X \in C$. Otherwise, M does not satisfy A, ($M \not\models A$).

A constraint $A = (X, C)$ is **consistent** if there exists M such that $M \models A$. Clearly, a constraint $A = (X, C)$ is consistent if and only if $C \neq \emptyset$.

We note that propositional atoms can be regarded as constraints. Let $a \in At$ and $M \subseteq At$. We define $C(a) = (\{a\}, \{\{a\}\})$. It is evident that $M \models C(a)$ if and only if $M \models a$. Therefore, in the thesis we often write a as a shorthand for the constraint $C(a)$. In fact, constraints also generalize weight atoms as well. For example, a weight atom $W = l[a_1 = w_1, \ldots, a_k = w_k]u$ is the constraint of the form (Y, Z), where $Y = \{a_1, \ldots, a_k\}$ and Z contains all subsets M of Y such that M satisfies W.

Constraints are building blocks of rules and programs. The following definition follows that in Marek and Truszczyński [102]

Definition 11. *A* **constraint rule** *is an expression of the following form:*

$$A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m) \tag{3.1}$$

where A, $A_1, \ldots, A_n$ *are constraints and* **not** *is the* **default negation** *operator. A* **constraint programs** *are sets of constraint rules.*

In the context of constraint programs, we refer to constraints and negated constraints as **literals**. Given a rule r of the form (3.1), the constraint (literal) A is the **head** of r and the set $\{A_1, \ldots, A_k, \ldots, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ of literals is the **body** of r[1]. We denote the head and the body of r by $hd(r)$ and $bd(r)$, respectively. We define the the **headset** of r, written $hset(r)$, as the domain of the head of r. That is, $hset(r) = Dom(hd(r))$.

For a constraint program P, we denote by $At(P)$ the set of atoms that appear in the domains of constraints in P. We define the **headset** of P, written $hset(P)$, as the union of the headsets of all rules in P.

The concept of satisfiability extends in a standard way to literals $\mathbf{not}(A)$ ($M \models \mathbf{not}(A)$ if $M \not\models A$), to sets (conjunctions) of literals and, finally, to constraint programs.

Definition 12. *Let* $M \subseteq At$ *be an interpretation. A rule (3.1) is* ***M-applicable*** *if* M *satisfies every literal in* $bd(r)$*. We denote by* $P(M)$ *the set of all* M*-applicable rules in* P*.*

Supportedness is a property of models. Intuitively, every atom a in a supported model must have "reasons" for being "in". Such reasons are M-applicable rules whose heads contain a in their domains. Formally, we have the following definition.

Definition 13. *Let* P *be a constraint program. A model* M *of* P *is* **supported** *if* $M \subseteq hset(P(M))$*.*

Definition 14. *Let* P *be a constraint program and* M *a set of atoms. A set* M' *is* **non-deterministically one-step provable** *from* M *by means of* P*, if* $M' \subseteq hset(P(M))$ *and* $M' \models hd(r)$*, for every rule* r *in* $P(M)$*.*

[1] As before, sometimes we view the body of a rule as the **conjunction** of its literals.

The **nondeterministic one-step provability operator** T_P^{nd} for a program P is an operator on $\mathcal{P}(At)$ such that for every $M \subseteq At$, $T_P^{nd}(M)$ consists of all sets that are nondeterministically one-step provable from M by means of P.

The operator T_P^{nd} is **nondeterministic** as it assigns to each $M \subseteq At$ a **family** of subsets of At, each being a possible outcome of applying P to M. In general, T_P^{nd} is **partial**, since there may be sets M such that $T_P^{nd}(M) = \emptyset$ (no set can be derived from M by means of P). For instance, if $P(M)$ contains a rule r such that $hd(r)$ is inconsistent, then $T_P^{nd}(M) = \emptyset$.

Now we introduce a special type of constraints.

Definition 15. *A constraint (X, C) is* **monotone** *if C is closed under superset, that is, for every $W, Y \subseteq X$, if $W \in C$ and $W \subseteq Y$ then $Y \in C$.*

W-atoms are examples of monotone constraints. For instance, the w-atom $W = l[a_1 = w_1, \ldots, a_k = w_k]$ is monotone. Indeed, let M be a model of W. That means $l \leq \sum_{a_i \in M} w_i$. Now we take an arbitrary $N \supseteq M$. If $a_i \in M$, then $a_i \in N$ as well. Since all w_i's are non-negative, $\sum_{a_i \in M} w_i \leq \sum_{a_i \in N} w_i$. Therefore, $l \leq \sum_{a_i \in N} w_i$. That is, $N \models W$ as well.

We call constraint programs built of monotone constraints **monotone-constraint programs** or **programs with monotone constraints**. That is, monotone-constraint programs consist of rules of the form

$$A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m) \tag{3.2}$$

where A, $A_1, \ldots, A_m$ are **monotone** constraints. If constraint A is inconsistent, we simplify the rule to the following one:

$$\leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)$$

From now on, unless explicitly stated otherwise, programs we consider are monotone-constraint programs.

3.1.1 Horn programs and bottom-up computations

Since we allow constraints with **infinite domains** and **inconsistent constraints** in heads of rules, the results given in this subsection are more general than their counterparts in the literature [97, 102]. Thus, for the sake of completeness, we present them with proofs.

A rule (3.2) is **Horn** if $k = m$ (no occurrences of the negation operator in the body or, equivalently, only monotone constraints). A constraint program is **Horn** if every rule in the program is Horn.

With a Horn constraint program we associate **bottom-up** computations, generalizing the corresponding notion of a bottom-up computation for a normal Horn program.

Definition 16. *Let P be a Horn program. A **P-computation** is a (transfinite) sequence $\langle X_\alpha \rangle$ such that*

1. $X_0 = \emptyset$,
2. *for every ordinal number α, $X_\alpha \subseteq X_{\alpha+1}$ and $X_{\alpha+1} \in T_P^{nd}(X_\alpha)$,*
3. *for every* **limit** *ordinal α, $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$.*

Let $t = \langle X_\alpha \rangle$ be a P-computation. Since for every $\beta < \beta'$, $X_\beta \subseteq X_{\beta'} \subseteq At$, there is a least ordinal number α_t such that $X_{\alpha_t} = X_\beta$ for all $\alpha_t < \beta$. In other words, there exists a least ordinal when the P-computation stabilizes. Indeed, if the P-computation never stabilizes, then the cardinality of X_α grows monotonically as α grows. Since P is fixed, therefore, $At(P)$ is fixed. Hence $|X_\alpha| \leq |At(P)|$ for all ordinal α. There is contradiction.

We refer to α_t as the **length** of the P-computation t.

Here is a simple example showing that some programs have computations of length exceeding ω and so transfinite induction in the definition cannot be avoided.

Example 10. *Let P be the program consisting of the following rules:*
$(\{a_0\}, \{\{a_0\}\}) \leftarrow .$
$(\{a_i\}, \{\{a_i\}\}) \leftarrow (X_{i-1}, \{X_{i-1}\})$, *for* $i = 1, 2, \ldots$
$(\{a\}, \{\{a\}\}) \leftarrow (X_\infty, \{X_\infty\})$,
where $X_i = \{a_0, \ldots a_i\}$, $0 \leq i$, and $X_\infty = \{a_0, a_1, \ldots\}$. Since the body of the last rule contains a constraint with an infinite domain X_∞, it does not become applicable in any finite step of computation. However, it does become applicable in the step ω and so $a \in X_{\omega+1}$. Consequently, $X_{\omega+1} \neq X_\omega$. $\triangle$

For a P-computation $t = \langle X_\alpha \rangle$, we call $\bigcup_\alpha X_\alpha$ the **result** of the computation and denote it by R_t. Directly from the definitions, it follows that $R_t = X_{\alpha_t}$.

Proposition 3. *Let P be a Horn constraint program and t a P-computation. Then R_t is a supported model of P.*

Proof. Let $M = R_t$ be the result of a P-computation $t = \langle X_\alpha \rangle$. We need to show that: (1) M is a model of P; and (2) $M \subseteq hset(P(M))$.

(1) Let us consider a rule $r \in P$ such that $M \models bd(r)$. Since $M = R_t = X_{\alpha_t}$ (where α_t is the length of t), $X_{\alpha_t} \models bd(r)$. Thus, $X_{\alpha_t+1} \models hd(r)$. Since $M = X_{\alpha_t+1}$, M is a model of r and, consequently, of P, as well.

(2) We prove by induction that, for every set X_α in the computation t, $X_\alpha \subseteq hset(P(M))$. The base case holds since $X_0 = \emptyset \subseteq hset(P(M))$.

If $\alpha = \beta + 1$, then $X_\alpha \in T_P^{nd}(X_\beta)$. It follows that $X_\alpha \subseteq hset(P(X_\beta))$. Since P is a Horn program and $X_\beta \subseteq M$, $hset(P(X_\beta)) \subseteq hset(P(M))$. Therefore, $X_\alpha \subseteq hset(P(M))$.

If α is a limit ordinal, then $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$. By the induction hypothesis, for every $\beta < \alpha$, $X_\beta \subseteq hset(P(M))$. Thus, $X_\alpha \subseteq hset(P(M))$. By induction, $M \subseteq hset(P(M))$. □

We use computations to define **derivable** models of Horn constraint programs.

Definition 17. *A set M of atoms is a* **derivable model** *of a Horn constraint program P if for some P-computation t, we have $M = R_t$.*

By Proposition 3, derivable models of P are supported models of P and, therefore, also models of P.

Since inconsistent monotone constraints may appear in the heads of Horn rules, there are Horn programs P and sets $X \subseteq At$, such that $T_P^{nd}(X) = \emptyset$. Thus, some Horn constraint programs have no computations and no derivable models. However, if a Horn constraint program has models, the existence of computations and derivable models is guaranteed.

To see this, let M be a model of a Horn constraint program P. We define a **canonical computation** $t^{P,M} = \langle X_\alpha^{P,M} \rangle$ by specifying the choice of the next set in the computation

in part (2) of Definition 16. Namely, for every ordinal β, we set

$$X^{P,M}_{\beta+1} = hset(P(X^{P,M}_{\beta})) \cap M.$$

That is, we include in $X^{P,M}_{\alpha}$ **all** those atoms occurring in the heads of $X^{P,M}_{\beta}$-applicable rules that belong to M. We denote the result of $t^{P,M}$ by $Can(P, M)$. Canonical computations are indeed P-computations.

Here is an example of the canonical computation:

Example 11. *Let P be the following Horn program:*
$(\{a, b\}, \{\{a\}, \{b\}, \{a, b\}\}) \leftarrow$
We can verify that $M = \{a\}$ is a model of P. Moreover, the canonical computation $t^{P,M} = \langle X^{P,M}_{\alpha}\rangle$, where $X_0 = \emptyset$, $X_\alpha = \{a\}$ for $\alpha > 1$. $\triangle$

Proposition 4. *Let P be a Horn constraint program. If $M \subseteq At$ is a model of P, the sequence $t^{P,M}$ is a P-computation.*

Proof. As P and M are fixed, to simplify the notation in the proof we write X_α instead of $X^{P,M}_{\alpha}$.

To prove the assertion, it suffices to show that for every ordinal α, (1) $hset(P(X_\alpha)) \cap M \in T^{nd}_P(X_\alpha)$, and (2) $X_\alpha \subseteq hset(P(X_\alpha)) \cap M$

(1) Let $X \subseteq M$ and $r \in P(X)$. Since all constraints in $bd(r)$ are monotone, and $X \models bd(r)$, $M \models bd(r)$, as well. From the fact that M is a model of P it follows now that $M \models hd(r)$. Consequently, $M \cap hset(P(X)) \models hd(r)$ for every $r \in P(X)$. Since $M \cap hset(P(X)) \subseteq hset(P(X))$,

$$M \cap hset(P(X)) \in T^{nd}_P(X).$$

Directly from the definition of the canonical computation for P and M we obtain that for every ordinal α, $X_\alpha \subseteq M$. Thus, (1), follows.

(2) We proceed by induction. The basis is evident as $X_0 = \emptyset$. Let us consider an ordinal $\alpha > 0$ and let us assume that (2) holds for every ordinal $\beta < \alpha$. If $\alpha = \beta + 1$, then

$X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M$. Thus, by the induction hypothesis, $X_\beta \subseteq X_\alpha$. Since P is a Horn constraint program, it follows that $P(X_\beta) \subseteq P(X_\alpha)$. Thus

$$X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M \subseteq hset(P(X_\alpha)) \cap M.$$

If α is a limit ordinal then for every $\beta < \alpha$, $X_\beta \subseteq X_\alpha$ and, as before, also $P(X_\beta) \subseteq P(X_\alpha)$. Thus, by the induction hypothesis for every $\beta < \alpha$,

$$X_\beta \subseteq hset(P(X_\beta)) \cap M \subseteq hset(P(X_\alpha)) \cap M,$$

which implies that

$$X_\alpha = \bigcup_{\beta<\alpha} X_\beta \subseteq hset(P(X_\alpha)) \cap M.$$

□

Canonical computations have the following **fixpoint** property.

Proposition 5. *Let P be a Horn constraint program. For every model M of P, we have $hset(P(Can(P, M))) \cap M = Can(P, M)$.*

Proof. Let α be the length of the canonical computation $t^{P,M}$. Then, $X^{P,M}_{\alpha+1} = X^{P,M}_\alpha = Can(P, M)$. Since $X_{\alpha+1} = hset(X_\alpha) \cap M$, the assertion follows. □

We now gather properties of derivable models that extend properties of the least model of normal Horn logic programs.

Proposition 6. *Let P be a Horn constraint program. Then:*

1. *For every model M of P, $Can(P, M)$ is a greatest derivable model of P contained in M*
2. *A model M of P is a derivable model if and only if $M = Can(P, M)$*
3. *If M is a minimal model of P then M is a derivable model of P.*

Proof. (1) Let M' be a derivable model of P such that $M' \subseteq M$. Let $T = \langle X_\alpha \rangle$ be a P-derivation such that $M' = R_t$. We want to prove that for every ordinal α, $X_\alpha \subseteq X_\alpha^{P,M}$. We proceed by transfinite induction. Since $X_0 = X_0^{P,M} = \emptyset$, the basis for the induction is evident. Let us consider an ordinal $\alpha > 0$ and assume that for every ordinal $\beta < \alpha$, $X_\beta \subseteq X_\beta^{P,M}$.

If $\alpha = \beta + 1$, then $X_\alpha \in T_P^{nd}(X_\beta)$ and so $X_\alpha \subseteq hset(P(X_\beta))$. By the induction hypothesis and by the monotonicity of the constraints in the bodies of rules in P, $X_\alpha \subseteq hset(P(X_\beta^{P,M}))$. Thus, since $X_\alpha \subseteq R_t = M' \subseteq M$,

$$X_\alpha \subseteq hset(P(X_\beta^{P,M})) \cap M = X_{\beta+1}^{P,M} = X_\alpha^{P,M}.$$

The case when α is a limit ordinal is straightforward as $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$ and $X_\alpha^{P,M} = \bigcup_{\beta<\alpha} X_\beta^{P,M}$.

(2) ($\Leftarrow$) If $M = Can(P, M)$, then M is the result of the canonical P-derivation for P and M. In particular, M is a derivable model of P.

($\Rightarrow$) if M is a derivable model of P, then M is also a model of P. From (1) it follows that $Can(P, M)$ is the greatest derivable model of P contained in M. Since M itself is derivable, $M = Can(P, M)$.

(3) From (1) it follows that $Can(P, M)$ is a derivable model of P and that $Can(P, M) \subseteq M$. Since M is a minimal model, $Can(P, M) = M$ and, by (2), M is a derivable model of P. □

3.1.2 Stable models

In this section, we recall and adapt the definition of stable models proposed in by Marek *et al.* [97, 102] to our monotone-constraint programs. Let P be a monotone-constraint program and M a subset of $At(P)$. The **reduct** of P, denoted by P^M, is a program obtained from P by:

1. removing from P all rules whose body contains a literal $\mathbf{not}(B)$ such that $M \models B$;
2. removing literals $\mathbf{not}(B)$ for the bodies of the remaining rules.

The reduct of a monotone-constraint program is Horn since it contains no occurrences of default negation. Therefore, the following definition is sound.

Definition 18. *Let P be a monotone-constraint program. A set of atoms M is a* **stable** *model of P if M is a derivable model of P^M. We denote the set of stable models of P by $St(P)$.*

The definitions of the reduct and stable models follow and generalize those proposed for normal logic programs, since in the setting of Horn constraint programs, derivable models play the role of a least model.

As in normal logic programming and its standard extensions, stable models of monotone-constraint programs are supported models and, consequently, models.

Proposition 7. *Let P be a monotone-constraint program. If $M \subseteq At(P)$ is a stable model of P, then M is a supported model of P.*

Proof. Let M be a stable model of P. Then, M is a derivable model of P^M and, by Proposition 3, M is a supported model of P^M. It follows that M is a model of P^M. Then directly from the definition of the reduct it follows that M is a model of P.

It also follows that $M \subseteq hset(P^M(M))$. For every rule r in $P^M(M)$, there is a rule r' in $P(M)$, which has the same head and the same non-negated literals in the body as r. Thus, $hset(P^M(M)) \subseteq hset(P(M))$ and, consequently, $M \subseteq hset(P(M))$. It follows that M is a supported model of P. □

If a normal logic program is Horn then its least model is its (only) stable model. Here we have an analogous situation.

Proposition 8. *Let P be a Horn monotone-constraint program. Then $M \subseteq At(P)$ is a derivable model of P if and only if M is a stable model of P.*

Proof. For every set M of atoms $P = P^M$. Thus, M is a derivable model of P if and only if it is a derivable model of P^M or, equivalently, a stable model of P. □

In the following sections, we show that several fundamental results concerning normal logic programs extend to the class of monotone-constraint programs.

3.2 Equivalence of *mac* programs

Program equivalence [47, 80, 79, 129] is an important concept due to its potential uses in program rewriting and optimization. Turner [129] presents an elegant characterization of strong equivalence of $lparse$-programs. Eiter and Fink [47] describe a similar characterization of uniform equivalence of normal and disjunctive logic programs. We show that both characterizations can be adapted to the case of monotone-constraint programs.

3.2.1 *M*-maximal models

A key role in our approach is played by models of Horn constraint programs satisfying a certain maximality condition.

Definition 19. *Let P be a Horn constraint program and let M be its model. A set $N \subseteq M$ such that N is a model of P and $M \cap hset(P(N)) \subseteq N$ is an **M-maximal** model of P, written $N \models_M P$.*

Intuitively, N is an M-maximal model of P if N satisfies each rule $r \in P(N)$ "maximally" with respect to M. That is, for every $r \in P(N)$, N contains all atoms in M that belong to $hset(r)$ — the domain of the head of r.

Example 12. *To illustrate this notion, let us consider a Horn constraint program P consisting of a single rule:*

$$1\{p, q, r\} \leftarrow 1\{s, t\}.$$

Let $M = \{p, q, s, t\}$ and $N = \{p, q, s\}$. One can verify that both M and N are models of P. Moreover, since the only rule in P is N-applicable, and $M \cap \{p, q, r\} \subseteq N$, N is an M-maximal model of P. On the other hand, $N' = \{p, s\}$ is not M-maximal even though N' is a model of P and it is contained in M. △

There are several similarities between properties of models of normal Horn programs and M-maximal models of Horn constraint programs. We state and prove here one of them that turns out to be especially relevant to our study of strong and uniform equivalence.

Proposition 9. *Let P be a Horn constraint program and let M be a model of P. Then M is an M-maximal model of P and $Can(P, M)$ is the least M-maximal model of P.*

Proof. The first claim follows directly from the definition. To prove the second one, we simplify the notation: we write N for $Can(P, M)$ and X_α for $X_\alpha^{P,M}$.

Let N' be any M-maximal model of P. We now show by transfinite induction that $N \subseteq N'$. Since $X_0 = \emptyset$, the basis for the induction holds. Let us consider an ordinal $\alpha > 0$ and let us assume that $X_\beta \subseteq N'$, for every $\beta < \alpha$.

Let us assume that $\alpha = \beta + 1$ for some $\beta < \alpha$. Then, since $X_\beta \subseteq N'$ and P is a Horn constraint program, we have $P(X_\beta) \subseteq P(N')$. Consequently,

$$X_\alpha = X_{\beta+1} = hset(P(X_\beta)) \cap M \subseteq hset(P(N')) \cap M \subseteq N',$$

the last inclusion follows from the fact thatN' is an M-maximal model of P.

If α is a limit ordinal, then $X_\alpha = \bigcup_{\beta<\alpha} X_\beta$ and the inclusion $X_\alpha \subseteq N'$ follows directly from the induction hypothesis.

To complete the proof, it is now enough to show that N is an M-maximal model of P. Clearly, $N \subseteq M$. Moreover, by Proposition 5, $hset(P(N)) \cap M = N$. Thus, N is indeed an M-maximal model of P. □

3.2.2 Strong equivalence and SE-models

Monotone-constraint programs P and Q are **strongly equivalent**, denoted by $P \equiv_s Q$, if for every monotone-constraint program R, $P \cup R$ and $Q \cup R$ have the same set of stable models.

To study the strong equivalence of monotone-constraint programs, we generalize the concept of an **SE-model** from Turner [129].

Definition 20. *Let P be a monotone-constraint program and let X, Y be sets of atoms. We say that (X, Y) is an* **SE-model** *of P if the following conditions hold: (1) $X \subseteq Y$; (2) $Y \models P$; and (3) $X \models_Y P^Y$ (that is, X is Y-maximal). We denote by $SE(P)$ the set of all SE-models of P.*

SE-models yield a simple characterization of strong equivalence of monotone-constraint programs. To state and prove this characterization, we need several auxiliary results.

Lemma 3.2.1. *Let P be a monotone-constraint program and let M be a model of P. Then (M, M) and $(Can(P^M, M), M)$ are both SE-models of P.*

Proof. The requirements (1) and (2) of an SE-model hold for (M, M). Furthermore, since M is a model of P, $M \models P^M$. Finally, we also have $hset(P(M)) \cap M \subseteq M$. Thus, $M \models_M P^M$.

Similarly, the definition of a canonical computation and Proposition 3, imply the first two requirements of the definition of SE-models for $(Can(P^M, M), M)$. The third requirement follows from Proposition 9. □

Lemma 3.2.2. *Let P and Q be two monotone-constraint programs such that $SE(P) = SE(Q)$. Then $St(P) = St(Q)$.*

Proof. If $M \in St(P)$, then M is a model of P and, by Lemma 3.2.1, $(M, M) \in SE(P)$. Hence, $(M, M) \in SE(Q)$ and, in particular, $M \models Q$. By Lemma 3.2.1 again,

$$(Can(Q^M, M), M) \in SE(Q).$$

By the assumption,

$$(Can(Q^M, M), M) \in SE(P)$$

and so $Can(Q^M, M) \models_M P^M$ or, in other terms, $Can(Q^M, M)$ is an M-maximal model of P^M. Since $M \in St(P)$, $M = Can(P^M, M)$. By Proposition 9, M is the least M-maximal model of P^M. Thus, $M \subseteq Can(Q^M, M)$. On the other hand, we have $Can(Q^M, M) \subseteq M$ and so $M = Can(Q^M, M)$. It follows that M is a stable model of Q. The other inclusion can be proved in the same way. □

Lemma 3.2.3. *Let P and R be two monotone-constraint programs. Then $SE(P \cup R) = SE(P) \cap SE(R)$.*

Proof. The assertion follows from the following two simple observations.

1. For every set Y of atoms, $Y \models (P \cup R)$ if and only if $Y \models P$ and $Y \models R$.

2. For every two sets X and Y of atoms, $X \models_Y (P \cup R)^Y$ if and only if $X \models_Y P^Y$ and $X \models_Y R^Y$.

For (1), we observe, from the definition of a model of a program, that

$$Y \models (P \cup R)$$

if and only if

$$Y \models r \text{ for every } r \in P \cup R$$

if and only if

$$Y \models r \text{ for every } r \in P$$

and

$$Y \models r' \text{ for every } r' \in R$$

if and only if

$$Y \models P \text{ and } Y \models R.$$

For (2), we assume X and Y are both models of $(P \cup R)^Y$ since, otherwise, (2) holds trivially.

$$X \models_Y (P \cup R)^Y$$

if and only if

$$Y \cap hset((P \cup R)^Y(X)) \subseteq X$$

if and only if

$$Y \cap hset(P^Y(X)) \subseteq X \text{ and } Y \cap hset(R^Y(X)) \subseteq X$$

if and only if

$$X \models_Y P^Y \text{ and } X \models_Y R^Y$$

□

Lemma 3.2.4. *Let P, Q be two monotone-constraint programs. If $P \equiv_s Q$, then P and Q have the same models.*

Proof. Let M be a model of P. By r we denote a constraint rule $(M, \{M\}) \leftarrow$. Then, $M \in St(P \cup \{r\})$. Since P and Q are strongly equivalent, $M \in St(Q \cup \{r\})$. It follows that M is a model of $Q \cup \{r\}$ and, therefore, also a model of Q. The converse inclusion can be proved in the same way. □

Theorem 13. *Let P and Q be monotone-constraint programs. Then $P \equiv_s Q$ if and only if $SE(P) = SE(Q)$.*

Proof. ($\Leftarrow$) Let R be an arbitrary monotone-constraint program. Lemma 3.2.3 implies that $SE(P \cup R) = SE(P) \cap SE(R)$ and $SE(Q \cup R) = SE(Q) \cap SE(R)$. Since $SE(P) = SE(Q)$, we have that $SE(P \cup R) = SE(Q \cup R)$. By Lemma 3.2.2, $P \cup R$ and $Q \cup R$ have the same stable models. Hence, $P \equiv_s Q$ holds.

($\Rightarrow$) Let us assume $SE(P) \setminus SE(Q) \neq \emptyset$ and let us consider $(X, Y) \in SE(P) \setminus SE(Q)$. It follows that $X \subseteq Y$ and $Y \models P$. By Lemma 3.2.4, $Y \models Q$. Since $(X, Y) \notin SE(Q)$, $X \not\models_Y Q^Y$. It follows that $X \not\models Q^Y$ or $hset(Q^Y(X)) \cap Y \not\subseteq X$. In the first case, there is a rule $r \in Q^Y(X)$ such that $X \not\models hd(r)$. Since $X \subseteq Y$ and Q^Y is a Horn constraint program, $r \in Q^Y(Y)$. Let us recall that $Y \models Q$ and so we also have $Y \models Q^Y$. It follows that $Y \models hd(r)$. Since $hset(r) \subseteq hset(Q^Y(X))$, $Y \cap hset(Q^Y(X)) \models hd(r)$. Thus, $hset(Q^Y(X)) \cap Y \not\subseteq X$ (otherwise, by the monotonicity of $hd(r)$, we would have $X \models hd(r)$).

The same property holds in the second case. Thus, it follows that $(hset(Q^Y(X)) \cap Y) \setminus X \neq \emptyset$. We define $X' = (hset(Q^Y(X)) \cap Y) \setminus X$.

Let R be a constraint program consisting of the following rules:

$$(x, \{x\}) \leftarrow$$
$$(y, \{y\}) \leftarrow (z, \{z\}),$$

for every $x \in X$, $y \in Y$, and $z \in X'$.

Let us consider a program $Q_0 = Q \cup R$. Since $Y \models Q$ and $X \subseteq Y$, $Y \models Q_0$. Thus, $Y \models Q_0^Y$ and, in particular, $Can(Q_0^Y, Y)$ is well defined. Since $R \subseteq Q_0^Y$, $X \subseteq Can(Q_0^Y, Y)$. Thus, the following holds.

$$hset(Q_0^Y(X)) \cap Y \subseteq hset(Q_0^Y(Can(Q_0^Y, Y))) \cap Y = Can(Q_0^Y, Y)$$

(the last equality follows from Proposition 5). We also have $Q \subseteq Q_0$ and so

$$X' \subseteq hset(Q^Y(X)) \cap Y \subseteq hset(Q_0^Y(X)) \cap Y.$$

Thus, $X' \subseteq Can(Q_0^Y, Y)$. Consequently, by Proposition 5 again, $Y \subseteq Can(Q_0^Y, Y)$. Since $Can(Q_0^Y, Y) \subseteq Y$, $Y = Can(Q_0^Y, Y)$ and so $Y \in St(Q_0)$.

Since P and Q are strongly equivalent, $Y \in St(P_0)$, where $P_0 = P \cup R$. Let us recall that $(X, Y) \in SE(P)$. By Proposition 9, $Can(P^Y, Y)$ is a least Y-maximal model of P^Y. Since X is a Y-maximal model of P (as $X \models_Y P^Y$), it follows that $Can(P^Y, Y) \subseteq X$. Since $X' \not\subseteq X$, $Can(P_0^Y, Y) \subseteq X$. Finally, since $X' \subseteq Y$, $Y \not\subseteq X$. Thus, $Y \neq Can(P_0^Y, Y)$, a contradiction.

It follows that $SE(P) \setminus SE(Q) = \emptyset$. By symmetry, $SE(Q) \setminus SE(P) = \emptyset$, too. Thus, $SE(P) = SE(Q)$. □

3.2.3 Uniform equivalence and UE-models

Let D be a set of atoms. By r_D we denote a monotone-constraint rule

$$r_D = \ (D, \{D\}) \leftarrow .$$

Adding a rule r_D to a program forces all atoms in D to be true (independently of the rest of the program).

Monotone-constraint programs P and Q are **uniformly equivalent**, denoted by $P \equiv_u Q$, if for every set of **atoms** D, $P \cup \{r_D\}$ and $Q \cup \{r_D\}$ have the same stable models.

An SE-model (X, Y) of a monotone-constraint program P is a **UE-model** of P if for every SE-model (X', Y) of P with $X \subseteq X'$, either $X = X'$ or $X' = Y$ holds. We write $UE(P)$ to denote the set of all UE-models of P. Our notion of a UE-model is a generalization of the notion of a UE-model from Eiter and Fink [47] to the setting of monotone-constraint programs.

We now present a characterization of uniform equivalence of monotone-constraint programs under the assumption that their sets of atoms are finite. One can prove a characterization of uniform equivalence of arbitrary monotone-constraint programs, generalizing one

of the results by Eiter and Fink [47]. However, both the characterization and its proof are more complex and, for brevity, we restrict our attention to the finite case only.

We start with an auxiliary result, which allows us to focus only on atoms in $At(P)$ when deciding whether a pair (X, Y) of sets of atoms is an SE-model of a monotone-constraint program P.

Lemma 3.2.5. *Let P be a monotone-constraint program, $X \subseteq Y$ two sets of atoms. Then $(X, Y) \in SE(P)$ if and only if $(X \cap At(P), Y \cap At(P)) \in SE(P)$.*

Proof. Since $X \subseteq Y$ is given, and $X \subseteq Y$ implies $X \cap At(P) \subseteq Y \cap At(P)$, the first condition of the definition of an SE-model holds on both sides of the equivalence.

Next, we note that for every constraint C, $Y \models C$ if and only if $Y \cap Dom(C) \models C$. Therefore, $Y \models P$ if and only if $Y \cap At(P) \models P$. That is, the second condition of the definition of an SE-model holds for (X, Y) if and only if it holds for $(X \cap At(P), Y \cap At(P))$.

Finally, we observe that $P^Y = P^{Y \cap At(P)}$ and $P(X) = P(X \cap At(P))$. Therefore,

$$Y \cap hset(P^Y(X)) = Y \cap hset(P^{Y \cap At(P)}(X \cap At(P))).$$

Since $hset(P^{Y \cap At(P)}(X \cap At(P))) \subseteq At(P)$, it follows that

$$Y \cap hset(P^Y(X)) \subseteq X$$

if and only if

$$Y \cap At(P) \cap hset(P^{Y \cap At(P)}(X \cap At(P))) \subseteq X \cap At(P).$$

Thus, $X \models_Y P^Y$ if and only if $X \cap At(P) \models_{Y \cap At(P)} P^{Y \cap At(P)}$. That is, the third condition of the definition of an SE-model holds for (X, Y) if and only if it holds for $(X \cap At(P), Y \cap At(P))$. □

Lemma 3.2.6. *Let P be a monotone-constraint program such that $At(P)$ is finite. Then for every $(X, Y) \in SE(P)$ such that $X \neq Y$, the set*

$$\{X' \colon X \subseteq X' \subseteq Y,\ X' \neq Y,\ (X', Y) \in SE(P)\} \tag{3.3}$$

has a maximal element.

Proof. If $At(P) \cap X = At(P) \cap Y$, then for every element $y \in Y \setminus X$, $Y \setminus \{y\}$ is a maximal element of the set (3.3). Indeed, since $(X, Y) \in SE(P)$, by Lemma 3.2.5, $(X \cap At(P), Y \cap At(P)) \in SE(P)$. Since $X \cap At(P) = Y \cap At(P)$ and $y \notin At(P)$, $X \cap At(P) = (Y \setminus \{y\}) \cap At(P)$. Therefore, $((Y \setminus \{y\}) \cap At(P), Y \cap At(P)) \in SE(P)$. Then from Lemma 3.2.5 and the fact $Y \setminus \{y\} \subseteq Y$, we have $(Y \setminus \{y\}, Y) \in SE(P)$. Therefore, $Y \setminus \{y\}$ belongs to the set (3.3) and so it is a maximal element of this set.

Thus, let us assume that $At(P) \cap X \neq At(P) \cap Y$. Let us define $X' = X \cup (Y \setminus At(P))$. Then $X \subseteq X' \subseteq Y$ and $X' \neq Y$. Moreover, no element in $X' \setminus X$ belongs to $At(P)$. That is, $X' \cap At(P) = X \cap At(P)$. Thus, by Lemma 3.2.5, $(X', Y) \in SE(P)$ and so X' belongs to the set (3.3). Since $Y \setminus X' \subseteq At(P)$, by the finiteness of $At(P)$ it follows that the set (3.3) contains a maximal element containing X'. In particular, it contains a maximal element. □

Theorem 14. *Let P and Q be two monotone-constraint programs such that $At(P) \cup At(Q)$ is finite. Then $P \equiv_u Q$ if and only if $UE(P) = UE(Q)$.*

Proof. ($\Leftarrow$) Let D be an arbitrary set of atoms and Y be a stable model of $P \cup \{r_D\}$. Then Y is a model of $P \cup \{r_D\}$. In particular, Y is a model of P and so $(Y, Y) \in UE(P)$. It follows that $(Y, Y) \in UE(Q)$, too. Thus, Y is a model of Q. Since Y is a model of r_D, $D \subseteq Y$. Consequently, Y is a model of $Q \cup \{r_D\}$ and thus, also of $(Q \cup \{r_D\})^Y$.

Let $X = Can((Q \cup \{r_D\})^Y, Y)$. Then $D \subseteq X \subseteq Y$ and, by Proposition 9, X is a Y-maximal model of $(Q \cup \{r_D\})^Y$. Consequently, X is a Y-maximal model of Q^Y. Since $X \subseteq Y$ and $Y \models Q$, $(X, Y) \in SE(Q)$.

Let us assume that $X \neq Y$. Then, by Lemma 3.2.6, there is a maximal set X' such that $X \subseteq X' \subseteq Y$, $X' \neq Y$ and $(X', Y) \in SE(Q)$. It follows that $(X', Y) \in UE(Q)$. Thus, $(X', Y) \in UE(P)$ and so $X' \models_Y P^Y$. Since $D \subseteq X'$, $X' \models_Y (P \cup \{r_D\})^Y$. We recall that Y is a stable model of $P \cup \{r_D\}$. Thus, $Y = Can((P \cup \{r_D\})^Y, Y)$. By Proposition 9, $Y \subseteq X'$ and so we get $X' = Y$, a contradiction. It follows that $X = Y$ and, consequently, Y is a stable model of $Q \cup \{r_D\}$.

By symmetry, every stable model of $Q \cup \{r_D\}$ is also a stable model of $P \cup \{r_D\}$.

($\Rightarrow$) First, we note that $(Y, Y) \in UE(P)$ if and only if Y is a model of P. Next, we note that P and Q have the same models. Indeed, the argument used in the proof of Lemma 3.2.4 works also under the assumption that $P \equiv_u Q$. Thus, $(Y, Y) \in UE(P)$ if and only if $(Y, Y) \in UE(Q)$.

Now let us assume that $UE(P) \neq UE(Q)$. Let (X, Y) be an element of $(UE(P) \setminus UE(Q)) \cup (UE(Q) \setminus UE(P))$. Without loss of generality, we can assume that $(X, Y) \in UE(P) \setminus UE(Q)$. Since $(X, Y) \in UE(P)$, it follows that

1. $X \subseteq Y$

2. $Y \models P$ and, consequently, $Y \models Q$

3. $X \neq Y$ (otherwise, by our earlier observations, (X, Y) would belong to $UE(Q)$).

Let $R = (Q \cup \{r_X\})^Y$. Clearly, R is a Horn constraint program. Moreover, since $Y \models Q$ and $X \subseteq Y, Y \models R$. Thus, $Can(R, Y)$ is defined. We have $X \subseteq Can(R, Y) \subseteq Y$. We claim that $Can(R, Y) \neq Y$. Let us assume to the contrary that $Can(R, Y) = Y$. Then $Y \in St(Q \cup \{r_X\})$. Hence, $Y \in St(P \cup \{r_X\})$, that is, $Y = Can((P \cup \{r_X\})^Y, Y)$. By Proposition 9, Y is the least Y-maximal model of $(P \cup \{r_X\})^Y$ and X is a Y-maximal model of $(P \cup \{r_X\})^Y$ (since $(X, Y) \in SE(P)$, $X \models_Y P^Y$ and so $X \models_Y (P \cup \{r_X\})^Y$, too). Consequently, $Y \subseteq X$ and, as $X \subseteq Y$, $X = Y$, a contradiction.

Thus, $Can(R, Y) \neq Y$. By Proposition 9, $Can(R, Y)$ is a Y-maximal model of R. Since $Q^Y \subseteq R$, it follows that $Can(R, Y)$ is a Y-maximal model of Q^Y and so $(Can(R, Y), Y) \in SE(Q)$. Since $Can(R, Y) \neq Y$, from Lemma 3.2.6 it follows that there is a maximal set X' such that $Can(R, Y) \subseteq X' \subseteq Y$, $X' \neq Y$ and $(X', Y) \in SE(Q)$. By the definition, $(X', Y) \in UE(Q)$. Since $(X, Y) \notin UE(Q)$. $X \neq X'$. Consequently, since $X \subseteq X'$, $X' \neq Y$ and $(X, Y) \in UE(P)$, $(X', Y) \notin UE(P)$.

Thus, $(X', Y) \in UE(Q) \setminus UE(P)$. By applying now the same argument as above to (X', Y) we show the existence of X'' such that $X' \subseteq X'' \subseteq Y$, $X' \neq X''$, $X'' \neq Y$ and $(X'', Y) \in SE(P)$. Consequently, we have $X \subseteq X''$, $X \neq X''$ and $Y \neq X''$, which contradicts the fact that $(X, Y) \in UE(P)$. It follows then that $UE(P) = UE(Q)$. $\square$

Example 15. *Let* $P = \{1\{p, q\} \leftarrow \mathbf{not}(2\{p, q\})\}$, *and* $Q = \{p \leftarrow \mathbf{not}(q),\ q \leftarrow \mathbf{not}(p)\}$. *Then* P *and* Q *are strongly equivalent. We note that both programs have* $\{p\}$, $\{q\}$, *and* $\{p, q\}$ *as models. Furthermore, we can verify that* $(\{p\}, \{p\})$, $(\{q\}, \{q\})$, $(\{p\}, \{p, q\})$, $(\{q\}, \{p, q\})$, $(\{p, q\}, \{p, q\})$ *and* $(\emptyset, \{p, q\})$ *are "all" SE-models of the two programs* [2].

Thus, by Theorem 13, P *and* Q *are strongly equivalent.*

We also observe that the first five SE-models are precisely UE-models of P *and* Q. *Therefore, by Theorem 14,* P *and* Q *are also uniformly equivalent.*

It is possible for two monotone-constraint programs to be uniformly but not strongly equivalent. If we add rule $p \leftarrow$ *to* P, *and rule* $p \leftarrow q$ *to* Q, *then the two resulting programs, say* P' *and* Q', *are uniformly equivalent. However, the two new programs are not strongly equivalent. The programs* $P' \cup \{q \leftarrow p\}$ *and* $Q' \cup \{q \leftarrow p\}$ *have different stable models. Another way to show that* P' *and* Q' *are not strongly equivalent is by observing that* $(\emptyset, \{p, q\})$ *is an SE-model of* Q' *but not an SE-model of* P'. △

3.3 From mac-programs to logic theories

3.3.1 Fages' Lemma for mac-programs

In general, supported models and stable models of a logic program (both in the normal case and the monotone-constraint case) do not coincide. Fages Lemma [50] (later extended by Erdem and Lifschitz [48]), establishes a sufficient condition under which a supported model of a normal logic program is stable. In this section, we show that Fages Lemma extends to programs with monotone constraints.

Definition 21. *A monotone-constraint program* P *is called* **tight** *on a set* $M \subseteq At(P)$ *of atoms if there exists a mapping* λ *from* M *to ordinals such that for every rule* $r = A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)$ *in* $P(M)$, *if* X *is the domain of* A *and* X_i *the domain of* A_i, $1 \leq i \leq k$, *then for every* $x \in M \cap X$ *and for every* $a \in M \cap \bigcup_{i=1}^{k} X_i$, $\lambda(a) < \lambda(x)$.

[2] From Lemma 3.2.5 and Theorem 13, it follows that only those SE-models that contain atoms only from $At(P) \cup At(Q)$ are the ones that decide if P and Q are strongly equivalent.

We now show that tightness provides a sufficient condition for a supported model to be stable. In order to prove a general result, we first establish it in the Horn case.

Lemma 3.3.1. *Let P be a Horn monotone-constraint program and let M be a supported model of P. If P is tight on M, then M is a stable model of P.*

Proof. Let M be an arbitrary supported model of P such that P is tight on M. Let λ be a mapping showing the tightness of P on M. We show that for every ordinal α and for every atom $x \in M$ such that $\lambda(x) \leq \alpha$, $x \in Can(P, M)$. We proceed by induction.

For the basis of the induction, let us consider an atom $x \in M$ such that $\lambda(x) = 0$. Since M is a supported model for P and $x \in M$, there exists a rule $r \in P(M)$ such that $x \in hset(r)$. Moreover, since P is tight on M, for every $A \in bd(r)$ and for every $y \in Dom(A) \cap M$, $\lambda(y) < \lambda(x) = 0$. Thus, for every $A \in bd(r)$, $Dom(A) \cap M = \emptyset$. Since $M \models bd(r)$ and since P is a Horn monotone-constraint program, it follows that $\emptyset \models bd(r)$. Consequently, $hset(r) \cap M \subseteq Can(P, M)$ and so $x \in Can(P, M)$.

Let us assume that the assertion holds for every ordinal $\beta < \alpha$ and let us consider $x \in M$ such that $\lambda(x) = \alpha$. As before, since M is a supported model of P, there exists a rule $r \in P(M)$ such that $x \in hset(r)$. By the assumption, P is tight on M and, consequently, for every $A \in bd(r)$ and for every $y \in Dom(A) \cap M$, $\lambda(y) < \lambda(x) = \alpha$. By the induction hypothesis, for every $A \in bd(r)$, $Dom(A) \cap M \subseteq Can(P, M)$. Since P is a Horn monotone-constraint program, $Can(P, M) \models bd(r)$. By Proposition 5, $hset(r) \cap M \subseteq Can(P, M)$ and so $x \in Can(P, M)$.

It follows that $M \subseteq Can(P, M)$. By the definition of a canonical computation, we have $Can(P, M) \subseteq M$. Thus, $M = Can(P, M)$. By Proposition 8, M is a stable model of P. □

Given this lemma, the general result follows easily.

Theorem 16. *Let P be a monotone-constraint program and let M be a supported model of P. If P is tight on M, then M is a stable model of P.*

Proof. One can check that if M is a supported model of P, then it is a supported model of the reduct P^M. Since P is tight on M, the reduct P^M is tight on M, too. Thus, M is

a stable model of P^M (by Lemma 3.3.1) and, consequently, a derivable model of P^M (by Proposition 8). It follows that M is a stable model of P. □

We give a tight mac-program in the following example.

Example 17. *Let P be the following mac-program:*

$(\{a,b\},\{\{a\},\{b\},\{a,b\}\}) \leftarrow (\{b\},\{\{b\}\})$

$(\{a\},\{\{a\}\}) \leftarrow.$

Let $M = \{a\}$. It is clear that M is a supported model of P: 1) M is a model of P; and 2) M is supported by the second rule. Moreover, we can check that P is tight on M. Indeed, we take $\lambda(a) = 0$, and $\lambda(b) = 1$. Then it is the mapping that satisfies the conditions in Definition 21. Therefore, by Theorem 16, M is a stable model of P.

Now by the definition of a stable model of an mac-program, we can also verify that M is indeed a stable model of P. △

However, Theorem 16 is only a sufficient condition of supported models being stable. Here is an example in which the program is not tight on a supported model M. Yet M is a stable model of the program.

Example 18. *Let P be the following mac-program:*

$A \leftarrow B$

$C \leftarrow$

$B \leftarrow$

where $A = (\{a,b\},\{\{a\},\{b\},\{a,b\}\})$, $B = (\{b\},\{\{b\}\})$, and $C = (\{a\},\{\{a\}\})$.

Let $M = \{a,b\}$. Then we can check that M is a supported model and a stable model of P. However, since atom b occurs both in $M \cap Dom(A)$ and $M \cap dom(B)$, no mappings could satisfy all the conditions in Definition 21 for the first rule. Therefore, P is **not** *tight on M.* △

3.3.2 Completion of mac-programs

A **completion** of a normal logic program [18] is a propositional theory whose models are precisely the supported models of the program. Thus, supported models of normal logic programs can be computed by means of SAT solvers. Under some conditions, for instance, when the assumptions of Fages Lemma hold, supported models are stable. Thus, computing models of the completion can yield stable models, an idea implemented in the first version of *cmodels* software [7].

Our goal is to extend the concept of the completion to programs with monotone constraints. The completion, as we define it, retains much of the structure of monotone-constraint rules. In this section we define the completion and prove a result relating supported models of programs to models of the completion. We discuss extensions of this result in the next section and their practical computational applications in Section 3.5.

To define the completion, we first introduce an extension of propositional logic with monotone constraints, a formalism we denote by PL^{mc}. A **formula** in the logic PL^{mc} is an expression built from monotone constraints by means of boolean connectives $\wedge$, $\vee$ and their **infinitary** counterparts (we show why we need infinitary conjunctions and disjunctions in a moment), $\rightarrow$ and $\neg$. The notion of a model of a constraint, which we discussed earlier, extends in a standard way to the class of formulas in the logic PL^{mc}. We use two symbols $\top$ and $\bot$ to denote PL^{mc} formulas that are satisfied by every truth assignment and that cannot be satisfied by any truth assignment, respectively.

For a set $L = \{A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m)\}$ of literals, we define

$$L^{\wedge} = A_1 \wedge \ldots \wedge A_k \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m.$$

Let P be a monotone-constraint program. We form the **completion** of P, denoted $Comp(P)$, as follows:

1. For every rule $r \in P$ we include in $Comp(P)$ a PL^{mc} formula

$$[bd(r)]^{\wedge} \rightarrow hd(r)$$

2. For every atom $x \in At(P)$, we include in $Comp(P)$ a PL^{mc} formula

$$(\{x\}, \{x\}) \rightarrow \bigvee\{[bd(r)]^{\wedge} : r \in P, x \in hset(r)\}$$

(When the set of rules in P is infinite, the disjunction may be infinitary).

The following theorem generalizes a fundamental result on the program completion from normal logic programming [18] to the case of programs with monotone constraints.

Theorem 19. *Let P be a monotone-constraint program. A set $M \subseteq At(P)$ is a supported model of P if and only if M is a model of $Comp(P)$.*

Proof. ($\Rightarrow$) Let us suppose that M is a supported model of P. Then M is a model of P, that is, for each rule $r \in P$, if $M \models bd(r)$ then $M \models hd(r)$. Since $M \models bd(r)$ if and only if $M \models [bd(r)]^{\wedge}$, it follows that all formulas in $Comp(P)$ of the first type are satisfied by M.

Moreover, since M is a supported model of P, $M \subseteq hset(P(M))$. That is, for every atom $x \in M$, there exists at least one rule r in P such that $x \in hset(r)$ and $M \models bd(r)$. Therefore, all formulas in $Comp(P)$ of the second type are satisfied by M, too.
($\Leftarrow$) Let us now suppose that M is a model of $Comp(P)$. Since $M \models bd(r)$ if and only if $M \models [bd(r)]^{\wedge}$, and since M satisfies formulas of the first type in $Comp(P)$, M is a model of P.

Let $x \in M$. Since M satisfies the formula $x \rightarrow \bigvee\{[bd(r)]^{\wedge} : r \in P, x \in hset(r)\}$, it follows that M satisfies $\bigvee\{[bd(r)]^{\wedge} : r \in P, x \in hset(r)\}$. That is, there is $r \in P$ such that M satisfies $[bd(r)]^{\wedge}$ (and so $bd(r)$, too) and $x \in hset(r)$. Thus, $x \in hset(P(M))$. Hence, M is a supported model of P. □

We observe that for the material in this section it is not necessary to require that constraints appearing in the bodies of program rules be monotone. However, since we are only interested in this case, we adopt the monotonicity assumption here, as well.

We now give an example of the completion of an mac-program.

Example 20. *Let P be the mac-program containing the following rules:*

$$A \leftarrow B, \mathbf{not}(C)$$

$$B \leftarrow D, \mathbf{not}(E)$$

where $A = (\{a, b, c\}, \{\{a, b\}, \{a, b, c\}\})$, $B = (\{a, b\}, \{\{a\}, \{b\}, \{a, b\}\})$, $C = (\{d, e\}, \{\{d, e\}\})$, $D = (\{b, d\}, \{\{b, d\}\})$, *and* $E = (\{c, d, e\}, \{\{c, d\}, \{c, e\}, \{c, d, e\}\})$.

Atom a *and* b *occur in the heads of both rules. Atom* c *only occurs in the head of the first rule. Atom* d *and* e *do not occur in the heads of either rule.*

Therefore, the completion $Comp(P)$ *contains the following* PL^{mc} *formulas:*

$$B \wedge \neg C \rightarrow A$$

$$D \wedge \neg E \rightarrow B$$

$$(\{a\}, \{a\}) \rightarrow (B \wedge \neg C) \vee (D \wedge \neg E)$$

$$(\{b\}, \{b\}) \rightarrow (B \wedge \neg C) \vee (D \wedge \neg E)$$

$$(\{c\}, \{c\}) \rightarrow (B \wedge \neg C)$$

$$(\{d\}, \{d\}) \rightarrow \bot$$

$$(\{e\}, \{e\}) \rightarrow \bot$$

Since d *and* e *do not occur in the heads of either rule of* P*, the disjunctions on the right-hand side of the last two formulas are empty. Since empty disjunctions cannot be satisfied by any truth assignment, we replace them with* $\bot$ *in those two formulas.* $\triangle$

3.3.3 Loop formulas for mac-programs

The completion alone is not quite satisfactory as it relates **supported**, not **stable**, models of monotone-constraint programs with models of PL^{mc} theories. Loop formulas, proposed by Lin and Zhao [81], provide a way to eliminate those supported models of normal logic programs that are not stable. Thus, they allow us to use SAT solvers to compute stable models of **arbitrary** normal logic programs and not only those for which supported and stable models coincide.

We now extend this idea to monotone-constraint programs. In this section, we restrict our considerations to programs P that are **finitary**, that is, $At(P)$ is finite. This restriction implies that monotone constraints that appear in finitary programs have finite domains.

Let P be a finitary monotone-constraint program. The **positive dependency graph** of P is the directed graph $G_P = (V, E)$, where $V = At(P)$ and $\langle u, v\rangle$ is an edge in E if there exists a rule $r \in P$ such that $u \in hset(r)$ and $v \in Dom(A)$ for some monotone constraint $A \in bd(r)$. We note that positive dependency graphs of finitary programs are finite.

Example 21. *Let P be the following mac-program:*

$(\{a, b\}, \{\{a\}, \{b\}, \{a, b\}\}) \leftarrow (\{b, c\}, \{\{b, c\}\})$

$(\{c\}, \{\{c\}\}) \leftarrow (\{a, d\}, \{\{a, d\}\})$

$(\{d\}, \{\{d\}\}) \leftarrow (\{e\}, \{\{e\}\})$

$(\{e\}, \{\emptyset, \{e\}\}) \leftarrow$

The positive dependency graph of P is shown in Figure 3.1. △

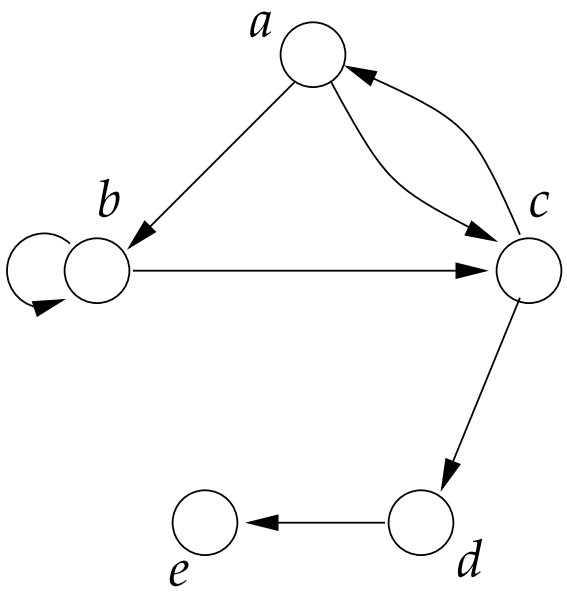

Figure 3.1: Positive dependency graph

Let $G = (V, E)$ be a directed graph. A set $L \subseteq V$ is a **loop** in G if the subgraph of G induced by L is strongly connected. A loop is **maximal** if it is not a proper subset of any other loop in G. Thus, maximal loops are vertex sets of strongly connected components of G. A maximal loop is **terminating** if there is no edge in G from L to any other maximal loop.

Following Example 21, we observe that (b), (a, b, c), and (a, c) are the loops in the positive dependency graph of P. Loop (a, b, c) is a maximal and terminating loop as well.

These concepts can be extended to the case of programs. By a **loop** (**maximal loop**, **terminating loop**) of a monotone-constraint program P, we mean the loop (maximal loop, terminating loop) of the positive dependency graph G_P of P. We observe that every finitary monotone-constraint program P has a terminating loop, since G_P is finite.

Let $X \subseteq At(P)$. By $G_P[X]$ we denote the subgraph of G_P **induced** by X. We observe that if $X \neq \emptyset$ then every loop of $G_P[X]$ is a loop of G_P.

Let P be a monotone-constraint program P. For every model M of P (in particular, for every model M of $Comp(P)$), we define $M^- = M \setminus Can(P^M, M)$. Since M is a model of P, M is a model of P^M. Thus, $Can(P^M, M)$ is well defined and so is M^-.

Lemma 3.3.2. *Let P be a monotone-constraint program and M a model of $Comp(P)$. If $M^- \neq \emptyset$, then there is a terminating loop contained in $G_P[M^-]$.*

Proof. Let $a \in M^-$. Since $M^- \subseteq M$ and M is a supported model of P, there is a rule $r \in P(M)$ such that $a \in hset(r)$ and $M \models bd(r)$. Let r' be a rule obtained from r by removing from the body of r all negated constraints. Since $r \in P(M)$, $r' \in P^M$.

We have $a \in M \setminus Can(P^M, M)$. By Proposition 5, $Can(P^M, M) \not\models bd(r')$. Let $A \in bd(r')$ be such that $Can(P^M, M) \not\models A$. Since $M \models A$, $M \cap Dom(A) \neq Dom(A) \cap Can(P^M, M)$. Thus, $Dom(A) \cap M^- \neq \emptyset$ and so the outdegree of a in $G_P[M^-]$ is greater than 0.

Since a is an arbitrary element of M^- and since $G_P[M^-]$ has only finitely many vertices, there exists a loop in $G_P[M^-]$. By finiteness of $G_P[M^-]$ again, there also exists a terminating loop in $G_P[M^-]$. □

For every loop in the graph G_P we now define the corresponding loop formula. First, for a constraint $A = (X, C)$ and a set $L \subseteq At$, we set $A_{|L} = (X, \{Y \in C \colon Y \cap L = \emptyset\})$ and call $A_{|L}$ the **restriction** of A to L. Next, let r be a monotone-constraint rule, say

$$r = \quad A \leftarrow A_1, \ldots, A_k, \mathbf{not}(A_{k+1}), \ldots, \mathbf{not}(A_m).$$

If $L \subseteq At$, then define a PL^{mc} formula $\beta_L(r)$ by setting

$$\beta_L(r) = A_{1|L} \wedge \ldots \wedge A_{k|L} \wedge \neg A_{k+1} \wedge \ldots \wedge \neg A_m.$$

Let L be a loop of a monotone-constraint program P. Then, the **loop formula** for L, denoted by $LP(L)$, is the PL^{mc} formula

$$LP(L) = \bigvee\{C(a) \colon a \in L\} \rightarrow \bigvee\{\beta_L(r) \colon r \in P \text{ and } L \cap hset(r) \neq \emptyset\}$$

(we recall that we use the convention to write a for the constraint $C(a) = (\{a\}, \{\{a\}\})$. A **loop completion** of a finitary monotone-constraint program P is the PL^{mc} theory

$$LComp(P) = Comp(P) \cup \{LP(L) \colon L \text{ is a loop in } G_P\}.$$

In Example 21, we take $M = \{a, b, c, d, e\}$. Clearly M is a model of P. $Can(P^M, M) = \{d, e\}$. Therefore, $G_P[M^-]$ is not empty. Moreover, (a, b, c) forms a terminating loop in $G_P[M^-]$. The loop formula for $L = \{a, b, c\}$ is

$$LP(L) = (C(a) \vee C(b) \vee C(c)) \rightarrow ((\emptyset, \{\emptyset\}) \vee (\{d\}, \{\emptyset\}))$$

where $C(a)$, $C(b)$, and $C(c)$ in the formula denote constraints $(\{a\}, \{\{a\}\})$, $(\{b\}, \{\{b\}\})$, and $(\{c\}, \{\{c\}\})$ respectively. We observe that $((\emptyset, \{\emptyset\}) \vee (\{d\}, \{\emptyset\}))$ cannot be satisfied since both constraints are inconsistent. Therefore, in order to satisfy this formula, we must not satisfy $a \vee b \vee c$. Since any supported model of P that contains atoms a, b, and c is not stable, this loop formula actually excludes all such supported models. This observation motivates us to use loop formulas to filter out non-stable supported models of P.

Formally, we have the following theorem, which exploits the concept of a loop formula to provide a necessary and sufficient condition for a model being a stable model.

Theorem 22. *Let P be a finitary monotone-constraint program. A set $M \subseteq At(P)$ is a stable model of P if and only if M is a model of $LComp(P)$.*

Proof. ($\Rightarrow$) Let M be a stable model of P. Then M is a supported model of P and, by Theorem 19, $M \models Comp(P)$.

Let L be a loop in P. If $M \cap L = \emptyset$ then $M \models \beta_L(r)$. Thus, let us assume that $M \cap L \neq \emptyset$. Since M is a stable model of P, M is a derivable model of P^M, that is, $M = Can(P^M, M)$. Let $(X_n)_{n=0,1,\ldots}$ be the canonical P^M-derivation with respect to M (since we assume that P is finite and each constraint in P has a finite domain, P-derivations reach their results in finitely many steps). Since $Can(P^M, M) \cap L = M \cap L \neq \emptyset$, there is a smallest index n such that $X_n \cap L \neq \emptyset$. In particular, it follows that $n > 0$ (as $X_0 = \emptyset$) and $L \cap X_{n-1} = \emptyset$.

Since $X_n = hset(P(X_{n-1}) \cap M$ and $X_n \cap L \neq \emptyset$, there is a rule $r \in P^M(X_{n-1})$ such that $hset(r) \cap L \neq \emptyset$, that is, such that $L \cap hset(r)) \neq \emptyset$. Let r' be a rule in P, which contributes r to P^M. Then, for every literal $\mathbf{not}(A) \in bd(r')$, $M \models \mathbf{not}(A)$. Let $A \in bd(r')$. Then $A \in bd(r)$ and so $X_{n-1} \models A$. Since $X_{n-1} \cap L = \emptyset$, $X_{n-1} \models A_{|L}$, too, By the monotonicity of $A_{|L}$, $M \models A_{|L}$. Thus, $M \models \beta_L(r')$. Since $hset(r') \cap L \neq \emptyset$, $L \cap hset(r)) \neq \emptyset$ and so $M \models LP(L)$. Thus, $M \models LComp(P)$.

($\Leftarrow$) Let us consider a set $M \subseteq At(P)$ such that M is not a stable model of P. If M is not a supported model of P that $M \not\models Comp(P)$ and so M is not a model of $LComp(P)$. Thus, let us assume that M is a supported model of P. It follows that $M^- \neq \emptyset$. Let $L \subseteq M^-$ be a terminating loop for $G_P[M^-]$.

Let r' be an arbitrary rule in P such that $L \cap hset(r')) \neq \emptyset$, and let r be the rule obtained from r' by removing negated constraints from its body. Now, let us assume that $M \models \beta_{r'}(L)$. It follows that for every literal $\mathbf{not}(A) \in bd(r')$, $M \models \mathbf{not}(A)$. Thus, $r \in P^M$. Moreover, since L is a terminating loop for $G_P[M^-]$, for every constraint $A \in bd(r')$, $Dom(A) \cap M^- \subseteq L$. Since $M \models A_{|L}$, it follows that $Can(P^M, M) \models A$. Consequently, $hset(r') \cap L \subseteq hset(r') \cap M \subseteq Can(P^M, M)$ and so $L \cap Can(P^M, M) \neq \emptyset$, a contradiction. Thus, $M \not\models \bigvee\{\beta_{r'}(L) \colon r' \in P$ and $L \cap hset(r')) \neq \emptyset\}$. Since $M \models \bigvee L$, it follows that $M \not\models LP(L)$ and so $M \not\models LComp(P)$. □

The following result follows directly from the proof of Theorem 22 and provides us with a way to filter out specific non-stable supported models from $Comp(P)$.

Theorem 23. *Let P be a finitary monotone-constraint program and M a model of $Comp(P)$.*

If M^- is not empty, then M violates the loop formula of every terminating loop of $G_P[M^-]$.

Finally, we point out that, Theorem 22 does not hold when a program P contains infinitely many rules. Here is a counterexample:

Let P be the set of following rules:

$$(\{a_0\}, \{\{a_0\}\}) \leftarrow (\{a_1\}, \{\{a_1\}\})$$
$$(\{a_1\}, \{\{a_1\}\}) \leftarrow (\{a_2\}, \{\{a_2\}\})$$
$$\ldots$$
$$(\{a_n\}, \{\{a_n\}\}) \leftarrow (\{a_{n+1}\}, \{\{a_{n+1}\}\})$$
$$\ldots$$

Let $M = \{a_0, \ldots, a_n, \ldots\}$. Then M is a supported model of P. The only stable model of P is $\emptyset$. However, $M^- = M \setminus \emptyset$ does not contain any terminating loop. The problem arises because there is an infinite simple path in $G_P[M^-]$. Therefore, $G_P[M^-]$ does not have a sink, yet it does not have a terminating loop either.

3.4 Programs with convex constraints

We now discuss programs with convex constraints, which are closely related to programs with monotone constraints. Programs with convex constraints are of interest as they do not involve explicit occurrences of the default negation operator **not**, yet are as expressive as programs with monotone-constraints. Moreover, they directly subsume an essential fragment of the class of *lparse*-programs [125].

A constraint (X, C) is **convex** if for every $W, Y, Z \subseteq X$ such that $W \subseteq Y \subseteq Z$ and $W, Z \in C$, we have $Y \in C$. A constraint rule (3.1) built of convex constraints only is a **convex-constraint rule**. Similarly, a constraint program built of convex-constraint rules is a **convex-constraint program**.

The concept of a model discussed in Section 3.1 applies to convex-constraint programs. To define supported and stable models of convex-constraint programs, we view them as special programs with monotone constraints.

To this end, we define the **upward** and **downward closures** of a constraint $A = (X, C)$ to be constraints $A^+ = (X, C^+)$ and $A^- = (X, C^-)$, respectively, where

$$C^+ = \{Y \subseteq X : \text{for some } W \in C, W \subseteq Y\}, \text{ and}$$
$$C^- = \{Y \subseteq X : \text{for some } W \in C, Y \subseteq W\}.$$

We note that the constraint A^+ is monotone. We call a constraint (X, C) **antimonotone** if C is closed under subset, that is, for every $W, Y \subseteq X$, if $Y \in C$ and $W \subseteq Y$ then $W \in C$. It is clear that the constraint A^- is antimonotone.

The upward and downward closures allow us to represent any convex constraint as the "conjunction" of a monotone constraint and an antimonotone constraint.Namely, we have the following property of convex constraints.

Proposition 10. *A constraint (X, C) is convex if and only if $C = C^+ \cap C^-$.*

Proof. ($\Leftarrow$) Let us assume that $C = C^+ \cap C^-$and let us consider a set M such that $M' \subseteq M \subseteq M''$, where $M', M'' \in C$. it follows that $M' \in C^+$ and $M'' \in C^-$. Thus, $M \in C^+$ and $M \in C^-$. Consequently, $M \in C$, which implies that (X, C) is convex.
($\Rightarrow$) The definitions directly imply that $C \subseteq C^+$ and $C \subseteq C^-$. Thus, $C \subseteq C^+ \cap C^-$. Let us consider $M \in C^+ \cap C^-$. Then there are sets $M', M'' \in C$ such that $M' \subseteq M$ and $M \subseteq M''$. Since C is convex, $M \in C$. Thus, $C^+ \cap C^- \subseteq C$ and so $C = C^+ \cap C^-$. □

This proposition suggests an encoding of convex-constraint programs as monotone-constraint programs. To present it, we need more notation. For a constraint $A = (X, C)$, we call the constraint $(X, \overline{C})$, where $\overline{C} = \mathcal{P}(X) \setminus C$, the **dual constraint** for A. We denote it by $\overline{A}$. It is a direct consequence of the definitions that a constraint A is monotone if and only if its dual $\overline{A}$ is antimonotone.

Let C be a convex constraint. We set $mc(C) = \{C\}$ if C is monotone. We set $mc(C) = \{\mathbf{not}(\overline{C})\}$, if C is antimonotone. We define $mc(C) = \{C^+, \mathbf{not}(\overline{C^-})\}$, if C is neither monotone nor antimonotone. Clearly, C and $mc(C)$ have the same models.

Let P be a convex-constraint program. By $mc(P)$ we denote the program with monotone constraints obtained by replacing every rule r in P with a rule r' such that

$$hd(r') = hd(r)^+ \text{ and } bd(r') = \bigcup\{mc(A) : A \in bd(r)\}$$

and, if $hd(r)$ is **not** monotone, also with an additional rule r'' such that

$$hd(r'') = (\emptyset, \emptyset) \text{ and } bd(r'') = \{\overline{hd(r)^-}\} \cup bd(r').$$

By our observation above, all constraints appearing in rules of $mc(P)$ are indeed monotone, that is, $mc(P)$ is a program with monotone constraints.

It follows from Proposition 10 that M is a model of P if and only if M is a model of $mc(P)$. We extend this correspondence to other semantics by defining M to be a supported (stable) model of a convex-constraint program P if M is a supported (stable) model of $mc(P)$.

With these definitions, monotone-constraint programs are (almost) directly convex-constraint programs. Namely, we note that monotone and antimonotone constraints are convex. Next, we observe that if A is a monotone constraint, the expression $\mathbf{not}(A)$ has the same meaning as the antimonotone constraint $\overline{A}$ in the sense that for every interpretation M, $M \models \mathbf{not}(A)$ if and only if $M \models \overline{A}$.

Let P be a monotone-constraint program. By $cc(P)$ we denote the program obtained from P by replacing literals $\mathbf{not}(A)$ in the bodies of rules in P with constraints $\overline{A}$. One can show that programs P and $cc(P)$ have the same models, supported models and stable models. In fact, for every monotone-constraint program P we have $P = mc(cc(P))$.

Remark. Another consequence of our discussion is that we can eliminate the default negation operator from the syntax at the price of allowing antimonotone constraints and using antimonotone constraints as negated literals. □

Due to the correspondences established above, one can extend to convex-constraint programs all concepts and results we discussed earlier in the context of monotone-constraint programs. In many cases, we can state these concepts and results **directly** in the language of convex-constraints. The most important for us are the notions of the completion and loop formulas, as they lead to new algorithms for computing stable models of $lparse$-programs. Therefore, we now discuss them in some detail.

As we just mentioned, we could use $Comp(mc(P))$ as a definition of the completion $Comp(P)$ for a convex-constraint logic program P. Under this definition Theorems 25 ex-

tends to the case of convex-constraint programs. However, $Comp(mc(P))$ involves monotone constraints and their negations and **not** convex constraints that appear in P. Therefore, we propose another approach, which preserves convex constraints of P.

To this end, we first extend the logic PL^{mc} with convex constraints. In this extension, which we denote by PL^{cc} and refer to as the **propositional logic with convex-constraints**, formulas are boolean combinations of convex constraints. The semantics of such formulas is given by the notion of a model obtained by extending over boolean connectives the concept of a model of a convex constraint.

Thus, the only difference between the logic PL^{mc}, which we used to define the completion and loop completion for monotone-convex programs and the logic PL^{cc} is that the former uses monotone constraints as building blocks of formulas, whereas the latter is based on convex constraints. In fact, since monotone constraints are special convex constraints, the logic PL^{mc} is a fragment of the logic PL^{cc}.

Let P be a convex-constraint program. The completion of P, denoted by $Comp(P)$, is the following set of PL^{cc} formulas:

1. For every rule $r \in P$ we include in $Comp(P)$ a PL^{cc} formula

   $$[bd(r)]^{\wedge} \rightarrow hd(r)$$

   (as before, for a set of convex constraints L, $L^{\wedge}$ denotes the conjunction of the constraints in L)

2. For every atom $x \in At(P)$, we include in $Comp(P)$ a PL^{cc} formula

   $$x \rightarrow \bigvee\{[bd(r)]^{\wedge} \colon r \in P,\ x \in hset(r)\}$$

   (again, we note that when the set of rules in P is infinite, the disjunction may be infinitary).

One can now show the following theorem.

Theorem 24. *Let P be a convex-constraint program and let $M \subseteq At(P)$. Then M is a supported model of P if and only if M is a model of $Comp(P)$.*

Proof. (Sketch) By the definition, M is a supported model of P if and only if M is a supported model of $mc(P)$. It is a matter of routine checking that $Comp(mc(P))$ and $Comp(P)$ have the same models. Thus the assertion follows from Theorem 19. □

Next, we restrict attention to **finitary** convex-constraint programs, that is, programs with finite set of atoms, and extend to this class of programs the notions of the positive dependency graph and loops. To this end, we exploit its representation as a monotone-constraint program $mc(P)$. That is, we define the positive dependency graph, loops and loop formulas for P as the positive dependency graph, loops and loop formulas of $mc(P)$, respectively. In particular, L is a loop of P if and only if L is a loop of $mc(P)$ and the loop formula for L, with respect to a convex-constraint program P, is defined as the loop formula $LP(L)$ with respect to the program $mc(P)$[3]. We note that since loop formulas for monotone-constraint programs only modify non-negated literals in the bodies of rules and leave negated literals intact, there seems to be no simple way to extend the notion of a loop formula to the case of a convex-constraint program P without making references to $mc(P)$.

We now define a **loop completion** of a finitary convex-constraint program P as the PL^{cc} theory

$$LComp(P) = Comp(P) \cup \{LP(L) \colon L \text{ is a loop of } P\}.$$

We have the following theorem that provides a necessary and sufficient condition for a set of atoms to be a stable model of a convex-constraint program.

Theorem 25. *Let P be a finitary convex-constraint program. A set $M \subseteq At(P)$ is a stable model of P if and only if M is a model of $LComp(P)$.*

Proof. (Sketch) Since M is a stable model of P if and only of M is a stable model of $mc(P)$, Theorem 22 implies that M is a stable model of P if and only if M is a stable model of $LComp(mc(P))$. It is a matter of routine checking that $LComp(mc(P))$ and $LComp(P)$ have the same models. Thus, the result follows. □

[3]There is one minor simplification one might employ. For a monotone constraint A, $\neg A$ and $\overline{A}$ are equivalent and $\overline{A}$ is antimonotone and so convex. Thus, we can eliminate the operator $\neg$ from loop formulas of convex-constraint programs by writing $\overline{A}$ instead of $\neg A$.

In a similar way, Theorem 23 implies the following result for convex-constraint programs.

Theorem 26. *Let P be a finitary convex-constraint program and M a model of $Comp(P)$. If M^- is not empty, then M violates the loop formula of every terminating loop of $G_P[M^-]$.*

We emphasize that one could simply use $LComp(mc(P))$ as a definition of the loop completion for a convex-constraint logic program. However, our definition of the completion component of the loop completion retains the structure of constraints in a program P, which is important when using loop completion for computation of stable models, the topic we address in the next section.

3.5 Computing stable models of *lparse*-programs via PL^{wa} solvers

In this section, we show how to use the theoretical results we present in Section 3.3.2, 3.3.3 and 3.4 to design and implement a new system for computing stable models of logic programs with weight atoms. The idea came from the development of solvers such as *cmodels* [7] and *assat* [81]. The difference between our implementation and *cmodles/assat* is that, we do not compile away weight atoms from the program. Our approach is sound because we have established the relationship between convex constraint programs and the logic PL^{cc}. Furthermore, existing implementations that can deal with theories in some instantiation of the logic PL^{cc} make our approach feasible in practice.

We consider two instantiations of logic PL^{cc}: one extends propositional logic with weight atoms; the other is the logic of pseudo-boolean constraints that roots in integer programming.

3.5.1 *lparse*-programs as convex constraint programs

We define the semantics of a w-atom in Chapter 2. It follows that a w-atom $l[a_1 = w_1, \ldots, a_k = w_k]u$ can be identified with a constraint (X, C), where $X = \{a_1, \ldots, a_k\}$ and

$$C = \{Y \subseteq X \colon l \leq \sum\{w_i \colon a_i \in Y\} \leq u\}.$$

Since w_i's are all non-negative, (X, C) is convex. Therefore, weight atoms represent a class of convex constraints and $lparse$-programs syntactically are a class of programs with convex constraints.

Marek, Niemelä and Truszczyński [97] and Marek and Truszczyński [102] have showed that we can encode $lparse$-programs as programs with monotone constraints so that the concept of a stable model is preserved. The transformation used there coincides with the encoding mc described in the previous section, when we restrict the latter to $lparse$-programs. Thus, we have the following theorem.

Theorem 27. *Let P be an $lparse$-program. A set $M \subseteq At$ is a stable model of P according to Definition 9 if and only if M is a stable model of P according to Definition 18 (when P is viewed as a convex-constraint program).*

It follows that to compute stable models of $lparse$-programs we can use the results obtained earlier in this chapter, specifically the results on program completion and loop formulas for convex-constraint programs.

3.5.2 Propositional logic extended with weight constraints

In Section 3.4, we extend the notions of completion and loop formulas to convex-constraint programs. Both completion and loop formulas are formulas in logic PL^{cc}. When we instantiate convex-constraint programs to $lparse$-programs, correspondingly we need to define an instantiation of logic PL^{cc} where convex constraints are w-atoms. We refer to this logic the **propositional logic with weight atoms** (or PL^{wa}, for short).

While it can be given a more general treatment, in this thesis we focus only on a certain class of formulas and theories.

Definition 22. *A* **clause** *in logic PL^{wa} is an expression of the form*

$$A_1 \vee \ldots \vee A_n \tag{3.4}$$

where each A_i is a w-atom.

We point out that, even though we do not explicitly use negations in a clause, a c-atom $[a]0$ is equivalent to the negative literal $\neg a$.

A **theory** of the logic PL^{wa} is any set of clauses.

The notion of satisfiability extends in a standard way to clauses and theories. We write interchangingly "is a model of" and "satisfies". We also write $I \models E$, when I is a model of an atom, w-atom, clause or theory E.

We propose logic PL^{wa} not only because the completion and loop formulas of $lparse$-programs are given in this logic, but also because some problems can be naturally and concisely represented as theories in this logic. Allowing boolean combinations of w-atoms in this logic is particularly useful.

The following problem, a slight generalization of the dominating-set problem we have discussed in Chapter 1, illustrates the usefulness of PL^{wa}-clauses in modeling.

Weighted dominating-set problem. Let $G = (V, E)$ be a directed weighted graph, where each edge (x, y) has a weight $w_{x,y} \geq 0$. Given an integer w, a set $D \subseteq V$ of vertices of G is ***w*-dominating** for G if for every vertex $x \in V$ at least one of the conditions listed below holds.

1. $x \in D$

2. the sum of weights of edges "from x to D" is at least w:
   $w \leq \sum_{(x,y)\in E, y\in D} w_{x,y}$

3. the sum of weights of edges "from D to x" is at least w:
   $w \leq \sum_{(z,x)\in E, z\in D} w_{z,x}$.

The following PL^{wa}-theory encodes the problem of the existence of a w-dominating set with at most k vertices. In the encoding we use atoms in_x, $x \in V$, with the intended meaning: **vertex x is in a w-dominating set**. The clauses of the theory are:

1. $1\{in_x\} \vee W_1 \vee W_2$, for every $x \in V$, where
   $W_1 = w[in_y = w_{x,y} \colon (x, y) \in E]$, and

$W_2 = w[in_z = w_{z,x} \colon (z,x) \in E]$.

These clauses enforce the defining constraint for a w-dominating set.

2. $\{in_x \colon x \in V\}k$.

   This clause guarantees that a selected subset has at most k vertices. The constraint that forms it is a cardinality constraint.

Pseudo-boolean constraint logic

Closely related to logic PL^{wa} is the **pseudo-boolean constraint logic**, denoted by logic PB, which roots in integer programming and operation research.

A **pseudo-boolean** constraint (***PB*-constraint**, for short) is an integer-programming constraint of the form

$$w_1x_1 + \ldots + w_kx_k \quad op \quad u, \tag{3.5}$$

where x_i are integer variables, each with the domain $\{0,1\}$, w_i are integers, which we refer to as **weights**, and l and u are integers called the **bounds**. Possible operators for op are $\{\leq, \geq, =\}$.

An assignment v of 0s and 1s to $x_i's$ is a **model** of (or **satisfies**) the constraint (3.5) if $w_1v(x_1) + \ldots + w_kv(x_k) \quad op \quad u$ holds.

By establishing the correspondence between integer values 0 and 1 on the one hand, and truth values $\mathbf{f}$ and $\mathbf{t}$, respectively, on the other, we can view integer 0-1 variables as propositional atoms. Furthermore, we can view PB-constraints as representations of propositional formulas. Specifically, we say that a constraint (3.5) represents a propositional formula φ (built of the same variables x_i, but now interpreted as propositional atoms) if (3.5) and φ have the same models (modulo the correspondence between $\{0,1\}$ and $\{\mathbf{f},\mathbf{t}\}$). In particular, a PB-constraint

$$x_1 + \ldots + x_k - y_1 - \ldots - y_m \geq 1 - m$$

represents a propositional clause

$$x_1 \vee \ldots \vee x_k \vee \neg y_1 \vee \ldots \vee \neg y_m.$$

Thus, PB-constraints generalize clauses, and sets of PB-constraints generalize propositional CNF theories. We call a set of PB constraints a **pseudoboolean satisfiability** (or PB SAT) instance.

3.5.3 Transformation between PB-theories and PL^{wa}-theories

Both logic PL^{wa} and logic PB are our targets logic for the completion and loop formulas of $lparse$-programs. Logic PL^{wa} aligns better to $lparse$-programs since disjunctions of w-atoms are allowed. On the other hand, more solvers are developed for theories in logic PB. Therefore, it is useful to transform a PL^{wa}-theory into a PB-theory and vice versa.

In this section, we define transformations between PB and PL^{wa} theories. We set two goals for the transformations: 1) they must transform theories from one side to "equivalent" theories on the other side; and 2) they must not increase the size of theories "significantly".

For the second goal, formally, we mean, if T is a theory and $\tau(T)$ is its translation, then $|\tau(T)| = O(|T|^c)$ for some constant c. In other words, the size of the transformation is bounded by a polynomial in the size of the input theory. We define $|T|$ as the sum of length of all clauses/constraints in T.

Sometimes, in order to bound the growth of the size polynomially, we need to introduce auxiliary new atoms into $\tau(T)$. Therefore, by "equivalence", we mean the equivalence with respect to $At(T)$ That is, we discard the new atoms in models of $\tau(T)$ and use the projected models to define equivalence. Formally we have the following definition.

Definition 23. *Let T_1 and T_2 be two theories and A a set of atoms. We say T_1 and T_2 are* **logically equivalent with respect to** *A, denoted by $T_1 \equiv^A T_2$, if $\{M \cap A \colon M \models T_1\} = \{M \cap A \colon M \models T_2\}$.*

From PB theories to PL^{wa} theories

Given a PB-theory T, which consists of a set of PB-constraints of the form (3.5), we define a transformation $\tau(T)$, whose result is a PL^{wa}-theory that is equivalent to T with respect to $At(T)$. We note that we overload the variable symbols in the PB theory T so that they represent propositional atoms in $\tau(T)$.

We rewrite a PB-constraint of the form (3.5) as follows:

$$-w_1 \times a_1 + \ldots + -w_h \times a_h + w_{h+1} \times a_{h+1} + \ldots + w_k \times a_k \;\; op \;\; b, \tag{3.6}$$

where w_i's are positive integers, b is an integer, a_i's are propositional atoms, and op stands for an element of $\{\leq, <, >, \geq\}$.

We take every PB-constraint of the form (3.6) from T. Let $N = \sum_{i=1}^{h} w_i$ and $W = \sum_{i=1}^{k} w_i$. We add to $\tau(T)$ the following set of PL^{wa}-clauses:

$$l[\overline{a_1} = w_1, \ldots, \overline{a_k} = w_k]u, \tag{3.7}$$

where

$$l = \begin{cases} 0 & \text{if } op \in \{\leq, <\} \\ b+N & \text{if } op \in \{\geq\} \\ b+N+1 & \text{if } op \in \{>\} \end{cases} \quad u = \begin{cases} W & \text{if } op \in \{>, \geq\} \\ b+N & \text{if } op \in \{\leq\} \\ b+N-1 & \text{if } op \in \{<\} \end{cases}$$

and

$$1\{\overline{a_i}, a_i\}1 \tag{3.8}$$

for every $i = 1, \ldots, h$.

In the transformation, atoms $\overline{a_i}$'s are new atoms. The last set of cardinality atoms define these new atoms so that $\overline{a_i}$ is true if and only if a_i is false. In other words, $\overline{ai}$ represents the dual literal of a_i. With the help of the new atoms, we can remove all negative weights from the original PB-constraint. Indeed, we observe that

$$-w_1 \times a_1 + \ldots + -w_h \times a_h + w_{h+1} \times a_{h+1} + \ldots + w_k \times a_k \;\; op \;\; b$$

is equivalent to

$$w_1 \times (1 - a_1) + \ldots + w_h \times (1 - a_h) + w_{h+1} \times a_{h+1} + \ldots + w_k \times a_k \;\; op \;\; (b + \sum_{i=1}^{h} w_i).$$

Moreover, we can view $(1 - a_i)$ as a 0-1 variable whose value is always the dual of the value of a_i. Now we use $\overline{a_i}$, which are new variables that do not occur in T, to represent $(1 - a_i)$, we get the following PB-constraint, which again is equivalent to the original one:

$$w_1 \times \overline{a_1} + \ldots + w_h \times \overline{a_h} + w_{h+1} \times a_{h+1} + \ldots + w_k \times a_k \;\; op \;\; (b + \sum_{i=1}^{h} w_i).$$

Now all the weights in this PB-constraint are positive. It is clear that such a PB-constraint is equivalent to the w-atom

$$l[\overline{a_1} = w_1, \ldots, \overline{a_k} = w_k]u,$$

where l and u are defined in (3.7). Then (3.8) defines the new atoms $\overline{a_i}$ via a set of cardinality atoms.

Finally, we observe that, for each PB-constraint in T, we introduce $h + 1$ unit clauses that contain w-atoms in (3.7) and (3.8) into $\tau(T)$. Since h is bounded by $|T|$, the number of clauses in $\tau(T)$ that represent one PB-constraint in T is bounded by $|T|$. Thus, the overall size of $\tau(T)$ is bounded by $|T|^2$.

From the argument above, the correctness of the following theorem is evident.

Theorem 28. *Let T be a PB-theory. Then T and $\tau(T)$ are logically equivalent with respect to $At(T)$. Moreover, $|\tau(T)| = O(|T|^2)$. When the size of PB-constraints in T is bounded by a constant, then $|\tau(T)| = O(|T|)$.*

We note that we use the term "equivalent" loosely here because logic PL^{wa} and logic PB use different assignments to define their semantics: interpretations for logic PL^{wa} and value assignments for logic PB. However, we can easily transform an interpretation into a value assignment from variables to values 0 or 1 and vice versa: a variable receives value 1 if and only if the corresponding atom receives truth value $\mathtt{t}$. Therefore, we use the term "equivalent" based on this correspondence between interpretations and value assignments.

From PL^{wa} theories to PB theories

Given a PL^{wa}-theory T, we define the transformation $\delta(T)$, whose result is a PB theory that is equivalent to T with respect to $At(T)$, as follows. Again we overload propositional atom symbols to represent 0-1 variables in PB-constraints.

For every PL^{wa}-clause of the form (3.4) in T, where each A_i is a w-atom of the form $l_i[a_{i,1} = w_{i,1}, \ldots, a_{i,k_i} = w_{i,k_i}]u_i$, we include in $\delta(T)$ the following set of PB-constraints. We use $W_i = \sum_{j=1}^{k_i} w_{i,j}$ to denote the sum of all weights in the w-atom A_i.

$$x_1 + \cdots + x_n \geq 1 \tag{3.9}$$

$$-x_i + y_i \geq 0 \tag{3.10}$$

$$-x_i + z_i \geq 0 \tag{3.11}$$

$$-y_i + -z_i + x_i \geq -1 \tag{3.12}$$

$$-l_i \times y_i + w_{i,1} \times a_{i,1} + \cdots + w_{i,k_i} \times a_{i,k_i} \geq 0 \tag{3.13}$$

$$(l_i - 1 - W_i) \times y_i + w_{i,1} \times a_{i,1} + \cdots + w_{i,k_i} \times a_{i,k_i} \leq (l_i - 1) \tag{3.14}$$

$$(W_i - u_i) \times z_i + w_{i,1} \times a_{i,1} + \cdots + w_{i,k_i} \times a_{i,k_i} \leq W_i \tag{3.15}$$

$$(u_i + 1) \times z_i + w_{i,1} \times a_{i,1} + \cdots + w_{i,k_i} \times a_{i,k_i} \geq (u_i + 1) \tag{3.16}$$

for $i = 1, \ldots, n$

In the PB-constraints, x_i, y_i and z_i are auxiliary new atoms that do not occur in $At(T)$ (they also do not occur as auxiliary atoms for other PL^{wa}-clauses).

The intuition is that, we first introduce atoms x_i for w-atoms A_i in a PL^{wa}-clause of the form (3.4). Thus, clause (3.4) becomes the following one:

$$x_1 \vee \ldots \vee x_n$$

where x_i's are propositional atoms. It is clear that such a propositional logic clause is equivalent to the PB-constraint given in (3.9).

Next, we need to define the equivalence between x_i and A_i. It is realized through PB-constraints (3.10) to (3.16). To this end, we first introduce two new variables y_i and z_i for each w-atom A_i in clause (3.4). The intuitive meaning of y_i (and z_i) is to represent the lower bound (and upper bound) constraint of A_i. In other words, we establish the following equivalence:

$$x_i \equiv y_i \wedge z_i,$$

which translates to PB-constraints (3.10), (3.11), and (3.12).

Constraints (3.13) and (3.14) define the equivalence between y_i and the lower bound part of A_i. Constraint (3.13) becomes trivial (any valuation satisfies it) when y_i gets value 0. On the other hand, when y_i gets value 1, constraint (3.13) becomes

$$w_{i,1} \times a_{i,1} + \cdots + w_{i,k_i} \times a_{i,k_i} \geq l.$$

Therefore, (3.13) ensures the following relationship

$$y_i \Rightarrow l[a_{i,1} = w_{i,1}, \ldots, a_{i,k_i} = w_{i,k_i}].$$

Similarly, constraint (3.14) ensures that

$$l[a_{i,1} = w_{i,1}, \ldots, a_{i,k_i} = w_{i,k_i}] \Rightarrow y_i.$$

Thus, two together ensure that

$$y_i \equiv l[a_{i,1} = w_{i,1}, \ldots, a_{i,k_i} = w_{i,k_i}].$$

That is, y_i is equivalent to the lower bound part of A_i.

With similar reasoning, we see that z_i is equivalent to the upper bound part of A_i. Therefore, we capture the w-atom A_i by those PB-constraints.

Finally, let us take look at the size of $\delta(T)$. It is clear that the number of PB-constraints we have in $\delta(T)$ for each PL^{wa}-clause is $7 \times n + 1$. Since n is bounded by $|T|$ (n is bounded by the size of the longest clause in T), our translation is bounded by $|T|^2$.

With the argument above, we prove the following theorem.

Theorem 29. *Let T be a PL^{wa}-theory. Then T and $\delta(T)$ are logically equivalent with respect to $At(T)$. Moreover, $|\delta(T)| = O(|T|^2)$. When the size of clauses in T is bounded by a constant, then $|\delta(T)| = O(|T|)$.*

3.5.4 Computing stable models of *lparse*-programs

In this section we present an algorithm for computing stable models of *lparse*-programs.

We have shown that *lparse*-programs are convex constraint programs and their completions and loop formulas are theories in logic PL^{wa} or logic PB. We now combine all these

theoretical results and create a new solver that computes stable models of $lparse$-programs via PL^{wa} or PB SAT solvers.

The SLS solvers that we present in Chapter 4 fit in this task well. Moreover, many off-the-shelf PB SAT solvers can be applied as well [1, 45, 82, 96, 132].

The algorithm of *pbmodels*

We follow the approach proposed by Lin and Zhao [81]. As in that paper, we first compute the completion of an $lparse$-program. Then, we iteratively compute models of the completion using a PB solver. Whenever a model is found, we test it for stability. If the model is not a stable model of the program, we extend the completion by loop formulas identified in Corollary 26. Often, adding a single loop formula filters out several models of $Comp(P)$ that are not stable models of P.

The work-flow is shown in Figure 3.2

The results given in the previous section ensure that our algorithm is correct. We present it in Figure 3.3. We note that it may happen that in the worst case we need exponentially many loop formulas [81] before we find the first stable model or we determine that no stable models exist. However, that problem arises only rarely in practical situations[4].

The implementation of $pbmodels$ supports several PB solvers including **satzoo** [45], **pbs** [1], **wsatoip** [132]. It also supports a program **wsatcc** [82] for computing models of PL^{wa}-theories. When this last program is used, the transformation, from "clausal" PL^{wa}-theories to pseudo-boolean theories is not needed. The first two of these four programs are complete PB solvers. The latter two are local-search solvers based on **wsat** [123].

We output the message "no stable model found" in the first line of the loop and not simply "no stable models exist" since in the case when A is a local-search algorithm, failure to find a model of the completion (extended with loop formulas in iteration two and the subsequent ones) does not imply that no models exist.

[4] In fact, in many cases programs turn out to be tight with respect to their supported models. Therefore, supported models are stable and no loop formulas are necessary at all.

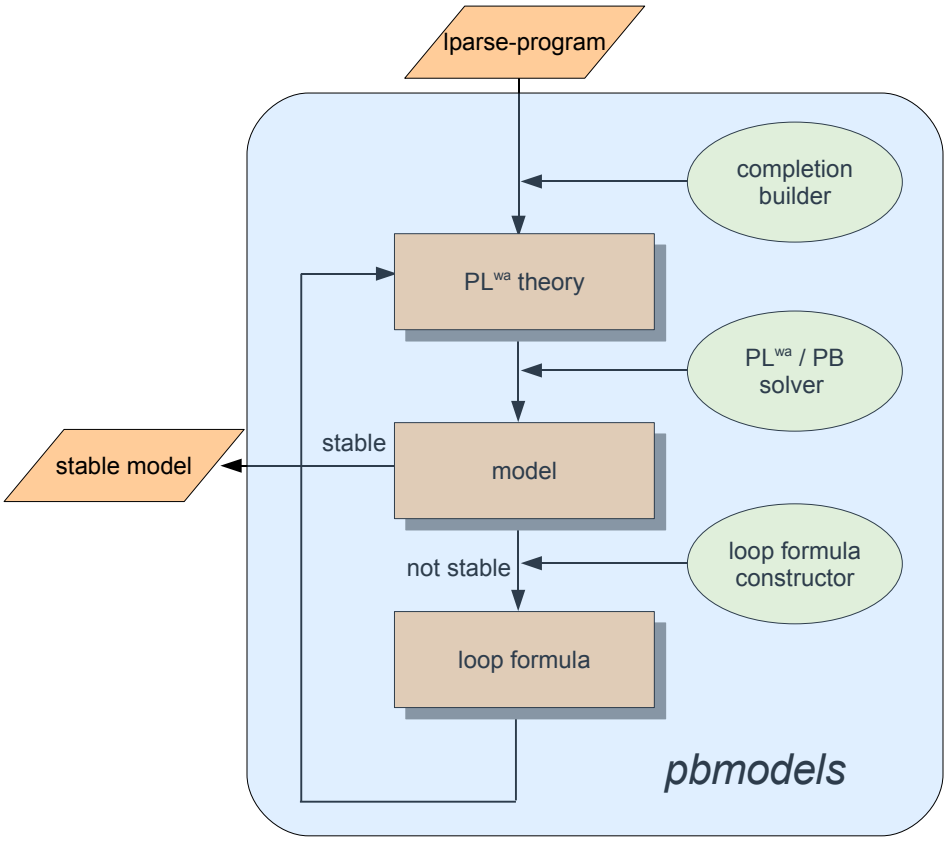

Figure 3.2: Work-flow of *pbmodels*

Input: P — an $lparse$-program;
A — a PB (or PL^{wa}) solver

BEGIN

1. compute the completion $Comp(P)$ of P;
2. convert $comp(P)$ to CNF theory T;
3. **if** A is a PB solver, convert T to PB-theory;
4. **do**
5. **if** (solver A finds no models of T) output "no stable models found" and terminate;
6. M := a model of T found by A;
7. **if** (M is stable) output M and terminate;
8. compute the reduct P^M of P with respect to M;
9. compute the greatest stable model M', contained in M, of P^M;
10. $M^- := M \setminus M'$;
11. find all terminating loops in M^-;
12. compute loop formulas and convert them into the set T' of PL^{wa}-clauses;
13. **if** A is a PB solver, convert T' into PB-constraints;
14. $T := T \cup T'$;
15. **while** (**true**);

END

Figure 3.3: Algorithm of $pbmodels$

Chapter 4

Stochastic Local Search in logic PL^{wa}-theories

In this chapter, we focus on algorithms that find satisfying truth assignments for a PL^{wa}-theory. To be precise, we focus on stochastic local search algorithms. This research was motivated by the results we obtained in Chapter 3, which lead us to a new solver that computes stable models of $lparse$-programs via PB SAT solvers. We observe that the completion formula and the loop formulas of an $lparse$-program are defined in logic PL^{wa}. In order to deploy PB SAT solvers to compute models of PL^{wa}-theories, we need to transform PL^{wa}-formulas into PB-theories, which are sets of PB-constraints. We use the transformation presented in Chapter 3. This transformation involves auxiliary new variables. In fact, to avoid size explosion, this type of transformation often introduces new variables as well as new clauses that correctly capture the connection between these new variables and the concepts they represent. The extra variables and clauses cause overhead to the underlying PB SAT solvers, especially local search solvers, to compute models. Therefore, we want to design solvers that compute models of logic PL^{wa}-theories directly. Moreover, because PB-constraints have been used to model search problems directly in the formalism of 0-1 integer programming and logic PL^{wa} subsumes this formalism, developing direct PL^{wa} solvers has its own interest.

This Chapter is organized as follows: Section 4.1 introduces the stochastic local search algorithm for theories in propositional logic. It is the basis on which our algorithms were developed. Section 4.2 presents our stochastic local search algorithms that directly target on PL^{wa} theories.

Our work present in this chapter appears in Proceedings of CP-03 [82] and AAAI-06 [87].

INPUT: T - a logic theory
OUTPUT: I - a satisfying assignment of T, or no output
BEGIN
1. **For** $i \leftarrow 1$ **to** $Max\text{-}Tries$, **do**
2. $I \leftarrow$ randomly generated truth assignment;
3. **For** $j \leftarrow 1$ **to** $Max\text{-}Flips$, **do**
4. **If** $I \models T$ **then return** I;
5. $a \leftarrow Heuristic(T, I)$;
6. $I \leftarrow Flip(T, I, a)$;
7. **End For** of j
8. **End For** of i

END

Figure 4.1: Algorithm $SLS\text{-}generic(T)$

4.1 Stochastic local search algorithm in propositional logic

In this section we describe a class of stochastic local-search (or SLS for short) algorithms designed to test satisfiability of theories in propositional logic. Stochastic local search was first proposed by Selman *et al.* [123] and has received much attention in SAT community because of its exceptional performance in solving some large satisfiable instances, which are beyond the power of solvers based on DPLL algorithms. One limitation of SLS solvers is that, they are **incomplete**, which means they may not be able to find a satisfying truth assignment if there is indeed one. We want to point out that SLS solvers are **sound**, which means whenever they report a satisfying truth assignment, it is a correct satisfying truth assignment.

Now we introduce the basic structure of algorithms behind SLS solvers. The algorithm executes $Max\text{-}Tries$ independent **tries**. Each try starts in a randomly generated complete truth assignment and consists of a sequence of up to $Max\text{-}Flips$ **flips**, that is, local changes to the current truth assignment. The algorithm terminates with a truth assignment that is a model of the input theory, or with no output at all (even though the input theory may in fact be satisfiable). We provide a detailed description of the algorithm $SLS\text{-}generic$ in Figure 4.1.

The input T is a theory in some logic. In this section, we assume T is a theory in propositional logic.

We point out that in the algorithm we use several parameters that, in the implementations, we enter from the command line. They are $Max\text{-}Tries$ and $Max\text{-}Flips$. All these parameters affect the performance of the program. We come back to this matter later in Chapter 5.

The procedure $Heuristic$ picks an atom from the theory. A typical heuristic function has a greedy part and a random part. It uses some probability to determine whether to follow the greedy part or the random part. It has been shown that the existence of a random move is essential for the local search solvers to escape from the local minima.

The procedure $Flip$ then flips the truth assignment to the atom returned from $Heuristic$. We note that the procedure $Flip$ may, in general, depend on the input theory T. It is not the case in $wsat$ and other similar algorithms but it is so in one of the algorithms we propose later. Thus, we include T as one of the arguments of the procedure $Flip$.

To obtain a concrete implementation of the algorithm $SLS\text{-}generic$, the key is to define a **heuristic**. Later in Section 4.2, we show that we can also extend the meaning of a **flip**. In the following, we first introduce several implementations of $SLS\text{-}generic$ that have been proposed in the literature for CNF theories. They are categorized into two major families. Namely, the **gsat** family and the **walksat** ($wsat$ for short) family. They differ from each other only in how they pick the atom to flip.

4.1.1 *Gsat* family

Gsat

Gsat was the first successful local search algorithm [123]. $Gsat$ uses only a greedy strategy to choose the atom to flip. Formally, $gsat$ uses the procedure given in Figure 4.2.

Intuitively, when $\Delta w(x) < 0$ (see Figure 4.2 for the definition of $\Delta w(x)$), the smaller the value of $\Delta w(x)$ is, the "better" the atom is, in a sense that flipping this atom improves the overall "degree" of satisfiability of the theory. This is a pure greedy strategy. The drawback of using such a strategy is that the algorithm may be trapped in local minima

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. **For** each atom x in T, **do**
2. $u_1 \leftarrow$ current number of unsat clauses;
3. $u_2 \leftarrow$ the number of unsat clauses after flipping x;
4. $\Delta w(x) \leftarrow u_2 - u_1$;
5. **End For**
6. **If** $minarg_x(\{\Delta w(x) : \Delta w(x) \leq 0\})$ exits **then**
7. return a random x from $minarg_x(\{\Delta w(x) : \Delta w(x) \leq 0\})$;
8. **Else**
9. halt;
10. **End If**

END

Figure 4.2: Algorithm $Heuristic\text{-}gsat(T, I)$

(that is, the situation in which there is no x such that $\Delta w(x) < 0$) and not be able to get out. In $gsat$, even if we use sideways moves when $\Delta w(x) = 0$ is the greatest decrease, **gsat** may still be stuck on a plateau.

In order to escape from local minima or plateaus, researchers have proposed several randomized techniques in $Heuristic$ procedure.

Simulated annealing (gsat-sa)

Simulated annealing introduces uphill moves when the algorithm ends up in local minima or plateaus.

In Figure 4.3, t is a parameter usually called *temperature.* The temperature can either be a constant or slowly decrease from a high one to near zero according to a cooling schedule.

Random walk (gsat-rw)

In simulated annealing, the random uphill move only depends on the probability $e^{-\Delta w(x)/t}$. Arbitrary atoms could be chosen if they have positive change in the number of unsatisfied clauses. With random walk, however, random moves are closely related to the atoms that appear in some unsatisfied clause. Given the property of the CNF formula that any unsat-

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. $x \leftarrow$ randomly chosen atom from T;
2. Compute $\Delta w(x)$ as it is done in $Heuristic\text{-}gsat$;
3. **If** $\Delta w(x) \leq 0$, **then**
4. return x;
5. **Else**
6. with probability $e^{-\Delta w(x)/t}$, return x;
7. with probability $1 - e^{-\Delta w(x)/t}$, goto step 1;
8. **End If**

END

Figure 4.3: Algorithm $Heuristic\text{-}gsat\text{-}sa(T, I)$

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. With probability p, return a randomly chosen atom occurring in some unsat clause;
2. With probability $1 - p$, follow $Heuristic\text{-}gsat$;

END

Figure 4.4: Algorithm $Heuristic\text{-}gsat\text{-}rw(T, I)$

isfied clause can be fixed by flipping a single arbitrary literal in that clause, such a random walk at least fixes one clause. The random walk strategy is given in Figure 4.4.

Random walk with TABU (gsat-rwtabu)

In both the basic GSAT algorithm or the simulated annealing algorithm, we may observe the oscillating phenomenon. That is, after flipping one atom, in the next iteration, the algorithm may flip it back.

For example,

Example 30. *Let T be a CNF theory consisting of the following eight clauses:*

$$a \vee b \vee c \tag{4.1}$$

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. Repeatedly call $Heuristic\text{-}gsat\text{-}rw(T, I)$ until the returned atom a is not in $TABU$;
2. Add a to $TABU$;
3. return a;

END

Figure 4.5: Algorithm $Heuristic\text{-}gsat\text{-}rwtabu(T, I)$

$$a \vee \neg b \tag{4.2}$$

$$a \vee \neg c \tag{4.3}$$

$$a \vee c \tag{4.4}$$

$$b \vee \neg a \tag{4.5}$$

$$b \vee \neg c \tag{4.6}$$

$$b \vee c \tag{4.7}$$

$$\neg a \vee \neg b \tag{4.8}$$

Let $I = \{a, b, c\}$. Truth assignment I does not satisfy T since it does not satisfy clause (4.8). We can verify that $\Delta(a) = \Delta(b) = 1$ and $\Delta(c) = 0$. Therefore, both **gsat** *and* **gsat-sa** *choose atom c to flip.*

Then we get a new truth assignment $I' = \{a, b\}$. In this case, we can compute that $\Delta(a) = \Delta(b) = 1$ and $\Delta(c) = 0$. Therefore, c will be chosen to flip again. $\triangle$

We see from the example that the oscillation is common in **gsat** and **gsat-sa**.

Even though with random walk technique, we can escape from this oscillation, it would be better that it could be prevented explicitly. By using a **TABU**, a finite FIFO list that stores all recent atoms the algorithm has chosen, we can implement this idea. The algorithm is given in Figure 4.5

4.1.2 Wsat family

Gsat, **gsat-rw**, and **gsat-rwtabu** all suffer from a problem when making a greedy move. That is, they all have to evaluate $\Delta w(x)$ for every atom x that occurs in the given formula T. This is a time consuming procedure. Even though in **gsat-sa**, it does not examine every atom, it does not perform the best move every time either. Thus **gsat-sa**, in general, may require much more flips than the other algorithms in **gsat** family. Experimental study [123] has showed the performance of **gsat** family is not good enough.

Wsat family was introduced in order to solve the problem [123]. The main difference between **gsat** family and **wsat** family lies in the set of atoms that should be examined. In **wsat** family, the algorithms only examine a subset of atoms, instead of all the atoms. Thus, the performance is improved.

There are at least four different algorithms in **wsat** family:

Wsat-G

Wsat-G is a counterpart of **gsat**. While we add randomness into a greedy strategy in **gsat**, we add greediness into a random walk strategy. Figure 4.6 shows this algorithm.

Wsat-B

So far, all the introduced algorithms always use the difference between the number of unsatisfied clauses before the flip and after the flip as the weight function in the greedy strategy. However, it is not the only possible weight function that could be used. In **Wsat-B**, the algorithm only uses so called **break-count** as the weight function.

By **break-count** of an atom x, we mean the number of already satisfied clauses that will receive truth value **f**, if atom x is flipped. The idea of **Wsat-B** is that, among a set of atoms, we want to pick the one that has the least break-count. Figure 4.7 gives the pseudo-code for **Wsat-B**:

Wsat-SKC

Wsat-SKC was proposed by Selman *et al.* [123]. It is one of the most effective algorithms

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. $C \leftarrow$ randomly selected unsatisfied clause in T;
2. **For** each atom x in C, **do**
3. compute $\Delta w(x)$ as it is done in $Heuristic\text{-}gsat$;
4. **End For**
5. **If** $minarg_x\{\Delta w(x) : \Delta w(x) \leq 0\}$ exists, **then**
6. with probability p,
return a randomly chosen atom in $minarg_x\{\Delta w(x) : \Delta w(x) \leq 0\}$;
7. with probability $1 - p$, return a randomly chosen atom in C;
8. **Else**
9. return a randomly chosen atom in C;
10. **End If**

END

Figure 4.6: Algorithm $Heuristic\text{-}wsat\text{-}G(T, I)$

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. $C \leftarrow$randomly selected unsatisfied clause in T;
2. **For** each atom x in C, **do**
3. compute $break\text{-}count(x)$;
4. **End For**
5. With probability p, return a randomly chosen atom in $minarg_x\{break\text{-}count(x)\}$;
6. With probability $1 - p$, return a randomly chosen atom in C;

END

Figure 4.7: Algorithm $Heuristic\text{-}wsat\text{-}B(T, I)$

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. $C \leftarrow$randomly selected unsatisfied clause in T;
2. **For each** atom x **in** C, compute $break\text{-}count(x)$;
3. **If** any of these atoms has $break\text{-}count = 0$ **then**
4. randomly choose an atom from these atoms and return it;
5. **Else**
6. with probability p, return a randomly chosen atom in C;
7. with probability $1 - p$,
return randomly an atom x with minimum $break\text{-}count(x)$;
8. **End If**

END

Figure 4.8: Algorithm $Heuristic\text{-}wsat\text{-}SKC(T, I)$

in **wsat** family. Similar to **wsat-B**, **wsat-SKC** also relies on the notion of the break-count of an atom.

The heuristic function in **wsat-SKC** is given in Figure 4.8.

The difference between **wsat-B** and **wsat-SKC** is that, in **wsat-SKC**, we always perform a **free** move when there is one. By a **free** move we mean flipping an atom that has zero break-count. Intuitively, free move means we will not break (or unsatisfy) any already satisfied clauses by taking such a move. Moreover, since flipping such an atom will fix all the broken (or unsatisfied) clauses that contain this atom, we actually will have a very good move towards the goal, that is to satisfy all the clauses.

wsat-rnovelty+

Wsat-rnovelty+ is another effective algorithm in **wsat** family. It was introduced by Hoos [68]. **Wsat-rnovelty+** uses both the break- and the make-count values. The **make-count** of an atom denotes the number of clauses that will become satisfied if the atom's value is flipped to its dual value.

We present a pseudo-code description in Algorithm 4.9.

To help search escape loops, with probability wp the heuristics chooses a random atom

INPUT: T - a CNF theory
I - a truth assignment
OUTPUT: x - an atom chosen for flipping
BEGIN
1. $C \leftarrow$randomly selected unsatisfied clause in T;
2. With probability wp, return a random atom from C;
3. **For each** atom x **in** C, $w(x) \leftarrow break\text{-}count(x) - make\text{-}count(x)$;
4. $age_{max} \leftarrow$ the maximum age of atoms in C;
5. $best \leftarrow$ the list of atoms x with the least $w(x)$;
6. $second \leftarrow$ the list of atoms x with the second least $w(x)$;
7. $diff \leftarrow w(x) - w(y)$, where $x \in best$ and $y \in second$;
8. **If** $\exists a \in best$ such that its age $< age_{max}$, return a;
9. **If** $diff > 1$, **then**
10. with probability $min\{2 - 2p, 1\}$, return a random atom from $best$;
11. with probability $1 - min\{2 - 2p, 1\}$, return a random atom from $second$;
12. **End If**
13. With probability $max\{1 - 2p, 0\}$, return a random atom from $best$;
14. With probability $1 - max\{1 - 2p, 0\}$, return a random atom from $second$;

END

Figure 4.9: Algorithm $Heuristic\text{-}wsat\text{-}rnovelty+(T, I)$

from the input clause C to return as the next atom to flip [1]. Otherwise, the algorithm selects an atom to flip based on the **quality** of atoms (the quality of an atom is a difference between its virtual break- and make-counts), the **age** of atoms (the age of an atom is defined as the time, measured in flips, when the atom was last flipped; initially all atoms have age 0 and their ages are updated by the flip procedure each time they are flipped), and a probability p, which determines whether an atom with the best or the second best quality value is selected. Even though the role of the parameter p is different here than it is in the case of the **SKC** heuristics, we call this value the **noise ratio**, as well. We also note that if all atoms in C have the same value of w, then one is selected randomly.

Finally, one consideration in implementing these algorithms is that, it is time consuming to calculate the break-count and the make-count of an atom. Therefore, researchers often use cached break-count and make-count of an atom in the heuristic function. Initially, the break-count and the make-count of each atom is computed with respect to the initial truth assignment using their definitions and stored in the cache. Then, within each try, the cache will be updated each time after an atom x is flipped. To be precise, we update, in the cache, the two counts of an atom by their changes, $\Delta break\text{-}count(x)$ and $\Delta make\text{-}count(x)$, assuming x is the atom flipped.

4.2 Extending *wsat* algorithms to logic PL^{wa}

We have seen various stochastic local search algorithms in Section 4.1. The main difference between the algorithms introduced in Section 4.1 and the ones we propose is that, our algorithms work on logic PL^{wa}-theories, which contain w-atoms in clauses. Because of this difference, the concept of the break-count or the make-count of an atom, or even the **flip** procedure becomes more complicated.

In this section, we work on the generic SLS algorithm $SLS\text{-}generic$, assuming that the input theory T is in logic PL^{wa}. To instantiate $SLS\text{-}generic$, we, again, need to decide how procedure $Heuristic$ and $Flip$ work. We follow two basic directions. In the first of

[1] Hoos [68] set wp to 0.01. We do not perform an extensive study on how the value of wp affects the **wsat-rnovelty+** algorithm.

them, we use a simple notion of a flip, that is, we always flip just one atom. We introduce, however, a more complex concept of the break-count and make-count, which we call the **virtual break-count** and the **virtual make-count**. We implement two instantiations of $wsat(cc)$-$generic$: $wsat(wa)$-skc and $wsat(wa)$-rnp. In the second approach, in addition to the virtual counts, we introduce a more complex concept of a flip, which we call the **double-flip**. In the case when the input theory T has some special property, the computation of the two counts becomes straightforward and do not involve virtual counts computation at all. We implement one instantiation of $wsat(cc)$-$generic$ following this direction: $wsat(wa)$-df. We discuss the three implementations in detail in the following subsections. For the other possible instantiation using the $RNovelty+$ heuristic, our first implementation does not yield good results in the experiments. Therefore, we do not include it in our discussion.

4.2.1 Virtual break-count and make-count

These two concepts depend on a particular representation of a PL^{wa}-theory T as a **multiset** of propositional clauses, $cl(T)$. We allow repetitions of clauses in sets and repetitions of literals in clauses, as by doing so we simplify some calculations.

We recall that we view w-atoms as propositional formulas. Given a w-atom W, by T_W we denote a certain CNF formula (which we also view as a multiset of its clauses) such that T_W is logically equivalent to W. We will specify T_W later.

Let us consider a PL^{wa}-clause C of the form (3.4). We define $cl(C)$ to be the multiset of propositional clauses that are disjuncts in the CNF formula obtained by replacing in C each w-atom W_i with the CNF formula T_{W_i} and by applying the distributivity law. For a PL^{wa}-theory T we then set

$$cl(T) = \bigcup \{cl(C) \colon C \in T\}.$$

Let I be a truth assignment. We define the **virtual break**- and **make-counts** of an atom x in a PL^{wa}-theory T with respect to I as the break- and make-counts of x in $cl(T)$ with respect to I. We denote these two quantities as $break\text{-}count_T(x)$ and $make\text{-}count_T(x)$,

respectively (we again drop the reference to I from the notation, as I is always determined by the context). It follows that

$$\mathit{break\text{-}count}_T(x) = \mathit{break\text{-}count}_{cl(T)}(x) = \sum\{\mathit{break\text{-}count}_{cl(C)}(x) \colon C \in T\}.$$

Similarly,

$$\mathit{make\text{-}count}_T(x) = \mathit{make\text{-}count}_{cl(T)}(x) = \sum\{\mathit{make\text{-}count}_{cl(C)}(x) \colon C \in T\}.$$

We now estimate $\mathit{break\text{-}count}_{cl(C)}(x)$ and $\mathit{make\text{-}count}_{cl(C)}(x)$. To this end we need more notation. Let W be a w-atom, I an interpretation and x a propositional atom. By $I^{\bar{x}}$ we denote the truth assignment obtained from I by flipping the truth value of x. Next, we define three sets of clauses that are relevant for $\mathit{break\text{-}count}_{cl(C)}(x)$ and $\mathit{make\text{-}count}_{cl(C)}(x)$ (we once again omit I in the notation):

1. $E_W(x) =$ the set of clauses in T_W that are satisfied by I but not by $I^{\bar{x}}$
2. $F_W(x) =$ the set of clauses in T_W that are not satisfied by I but are satisfied by $I^{\bar{x}}$
3. $G_W(x) =$ the set of clauses in T_W that are not satisfied by I nor by $I^{\bar{x}}$.

We observe that $E_W(x) = F_W(x) = \emptyset$ if x does not appear in W. We set $e(x) = |E_W(x)|$, $f(x) = |F_W(x)|$ and $g(x) = |G_W(x)|$.

We now have the following theorem:

Theorem 31. *Let C be a PL^{wa}-clause of the form (3.4). Let cl be the transformation we define above. Let $e_i(x)$, $f_i(x)$, and $g_i(x)$'s be three cardinalities of the three sets $E_{W_i}(x)$, $F_{W_i}(x)$ and $G_{W_i}(x)$ with respect to an atom x and cl. Then we have the following equations:*

$$\mathit{break\text{-}count}_{cl(C)}(x) = \prod_{i=1}^{n}(e_i(x) + g_i(x)) - \prod_{i=1}^{n} g_i(x). \tag{4.9}$$

and

$$\mathit{make\text{-}count}_{cl(C)}(x) = \prod_{i=1}^{n}(f_i(x) + g_i(x)) - \prod_{i=1}^{n} g_i(x). \tag{4.10}$$

Proof. Equation (4.9): indeed, every clause in $cl(C)$ is of the form $D_1 \vee \ldots \vee D_n$, where $D_i \in T_{W_i}$, $1 \leq i \leq n$. For a clause in $cl(C)$ to get "unsatisfied" with the flip of x, all D_i's in $cl(C)$ must satisfy the following two conditions:

1. each D_i is chosen from $E_{W_i}(x) \cup G_{W_i}(x)$; and
2. at least one D_i is chosen from $E_{W_i}(x)$.

Since $E_{W_i}(x) \cap G_{W_i}(x) = \emptyset$, the number of clauses in $cl(C)$ such that each D_i is chosen from $E_{W_i}(x) \cup G_{W_i}(x)$ is given by

$$\prod_{i=1}^{n}(e_i(x) + g_i(x)).$$

Then among these clauses, the ones in which each D_i is chosen from $G_{W_i}(x)$ do not contribute to the break-count of x. The number of such clauses is given by

$$\prod_{i=1}^{n} g_i(x).$$

Then the equation (4.9) follows.

Equation (4.10) holds for the similar reason. □

Estimating e, f and g

To make formulas (4.9) and (4.10) complete, we need to specify a CNF representation T_W of a w-atom W and, given this representation and a truth assignment I, for each atom x find formulas for e, f and g (in this section, we omit x in $e(x)$, $f(x)$ and $g(x)$ as x always exists in the context).

Let us then consider a w-atom W:

$$W = l[a_1 = w_1, \ldots, a_k = w_k]u,$$

where all w_i are non-negative. For each atom a_i we introduce new atoms a_i^j, $1 \leq j \leq w_i$. We then define a cardinality constraint

$$W' = l[a_1^1, \ldots, a_1^{w_1}, \ldots, a_k^1, \ldots, a_k^{w_k}]u,$$

and a set of formulas

$$EQ = \{a_i \equiv a_i^j : 1 \leq i \leq k, 1 \leq j \leq w_i\}.$$

The w-atom W and $\{W'\} \cup EQ$ are equivalent in the following sense. There is a one-to-one correspondence between models of W and models of $\{W'\} \cup EQ$. The corresponding models coincide on the set $\{a_1, \ldots, a_k\}$. In the case of the theory $\{W'\} \cup EQ$, the part of the model contained in $\{a_1, \ldots, a_k\}$ determines the rest, as models of $\{W'\} \cup EQ$ must satisfy formulas in EQ.

Let set S consist of the following clauses:

$$\neg x_{i_1} \vee \ldots \vee \neg x_{i_{u+1}} \tag{4.11}$$

for every $(u+1)$-element subset $\{x_{i_1}, \ldots, x_{i_{u+1}}\}$ of $\{a_1^1, \ldots, a_1^{w_1}, \ldots, a_k^1, \ldots, a_k^{w_k}\}$, and

$$x_{i_1} \vee \ldots \vee x_{i_{K-l+1}} \tag{4.12}$$

for every $(K-l+1)$-element subset $\{x_{i_1}, \ldots, x_{i_{K-l+1}}\}$ of $\{a_1^1, \ldots, a_1^{w_1}, \ldots, a_k^1, \ldots, a_k^{w_k}\}$, where $K = \sum w_i$.

The following theorem says that the cardinality constraint W' is equivalent to S:

Theorem 32. *Let W' be the cardinality constraint and S the set of clauses we defined above. Then a truth assignment I satisfies W' if and only if I satisfies S.*

Proof. ($\Leftarrow$): let I be a truth assignment that satisfies S. We need to show that $I \models W'$ as well. Assume it is not the case. Then, either $|X| \leq l-1$ or $|X| \geq u+1$, where $X = \{a_i^j : I \models a_i^j\}$.

If it is the first case, let us take $X' = \{a_1^1, \ldots, a_k^{w_k}\} \setminus X$. It is clear that for every $a \in X'$, $I \not\models a$. Since $|X| \leq l-1$, $|X'| \geq K-l+1$. Therefore, there exists a $(K-l+1)$-element subset of X' that is also a $(K-l+1)$-element subset of $\{a_1^1, \ldots, a_k^{w_k}\}$. Since this subset forms a clause of the form (4.12), this clause is not satisfied by I. It is a contradiction.

If it is the second case, then $|X| \geq u+1$. Therefore, there exists a $(u+1)$-element subset of X that is also a $(u+1)$-element subset of $\{a_1^1, \ldots, a_k^{w_k}\}$. Then the clause formed by this subset (it is of form (4.11)) is not satisfied by I. Again, a contradiction.

Therefore, I must satisfy W'.

($\Rightarrow$): let $I \models W'$. Since the lower bound constraint of W' is satisfied by I, then the number of atoms from $\{a_1^1, \ldots, a_k^{w_k}\}$ that are false under I is at most $K - l$. Therefore, for any $K - l + 1$-element subset of $\{a_1^1, \ldots, a_k^{w_k}\}$, there exists at least one atom that is true. Therefore, all clauses of form (4.12) are satisfied.

Similarly, since the upper bound constraint of W' is satisfied by I, the number of atoms from $\{a_1^1, \ldots, a_k^{w_k}\}$ that are true under I is at most u. Therefore, for any $u + 1$-element subset of $\{a_1^1, \ldots, a_k^{w_k}\}$, there exists at least one atom that is false. Therefore, all clauses of form (4.11) are satisfied.

Therefore, I satisfies S. □

Thus, W is equivalent to $S \cup EQ$ (in the same sense as before). Consequently, W is equivalent (has the same models) as the multiset of clauses obtained from S by replacing each atom a_i^j with a_i. We define T_W to be this multiset. We also note that clauses in this multiset may contain multiple occurrences of the same literals.

We do not simplify T_W further (that is, we do not eliminate duplicate clauses nor duplicate occurrences of literals in clauses) since the multiset form of T_W makes it easier to compute the cardinalities e, f and g of the sub-multisets $E_{W,x}$, $F_{W,x}$ and $G_{W,x}$ of T_W and their cardinalities e, f and g. Namely, we have the following formulas for e, f and g:

$$e = \begin{cases} 0 & \text{case 1} \\ \binom{N+w}{K-l+1} - \binom{N}{K-l+1} & \text{case 2} \\ \binom{P+w}{u+1} - \binom{P}{u+1} & \text{otherwise} \end{cases} \tag{4.13}$$

$$f = \begin{cases} 0 & \text{case 1} \\ \binom{P}{u+1} - \binom{P-w}{u+1} & \text{case 2} \\ \binom{N}{K-l+1} - \binom{N-w}{K-l+1} & \text{otherwise} \end{cases} \tag{4.14}$$

$$g = \begin{cases} \binom{N}{K-l+1} + \binom{P}{u+1} & \text{case 1} \\ \binom{N}{K-l+1} + \binom{P-w}{u+1} & \text{case 2} \\ \binom{P}{u+1} + \binom{N-w}{K-l+1} & \text{otherwise.} \end{cases} \tag{4.15}$$

Case 1 covers all situations when x does not occur in W. Case 2 covers situations when x occurs in W and $I \models x$. In these formulas we use the notation $K = \sum w_i$, $P = \sum_{I \models a_i} w_i$, $N = \sum_{I \not\models a_i} w_i$, and write w for the weight of atom x in W (if x occurs in W).

We now provide arguments for each case of (4.13), (4.14), and (4.15). In the following, let I be a truth assignment. Let us assume $\mathcal{N} = \{a_i^j : 1 \leq i \leq k,\ 1 \leq j \leq w_i,\ I \not\models a_i\}$ and $\mathcal{P} = \{a_i^j : 1 \leq i \leq k,\ 1 \leq j \leq w_i,\ I \models a_i\}$. It is clear that $N = |\mathcal{N}|$ and $P = |\mathcal{N}|$.

Case 1. Since x does not occur in W, by the definition of sets $E_{W,x}$ and $F_{W,x}$, $e = f = 0$. Therefore, (4.13) and (4.14) are correct in this case. By the definition of $G_{W,x}$, all clauses in $G_{W,x}$ are those not satisfied by I and not satisfied by $I^{\bar{x}}$. Since x does not occur in W, all clauses that are not satisfied by I form precisely the set G. That is, a clause $C \in G_{W,x}$ if and only if

1. C is obtained from a clause C' in S of the form (4.11) such that C' contains only atoms from $\mathcal{P}$; or

2. C is obtained from a clause C' in S of the form (4.12) such that C' contains only atoms from $\mathcal{N}$.

For the first type of C, there are $\binom{P}{u+1}$ such clauses in S. For the second type of C, there are $\binom{N}{K-l+1}$ such clauses in S. Since we do not remove duplicate clauses when we generate T_W from S, the number of clauses C in $G_{W,x}$ is precisely $\binom{N}{K-l+1} + \binom{P}{u+1}$. Thus, case 1 of (4.15) holds.

Case 2. We first look at the case 2 of the equation (4.13). In this case, x occurs in W and $I \models x$. Let us assume that $x = a_1$. Since $I \models a_1$, $\{a_1^1, \ldots, a_1^{w_1}\} \cap \mathcal{N} = \emptyset$. The definitions of S and $E_{W,x}$ imply that $C \in E_{W,x}$ if and only if C is obtained from a clause C' in S of the form (4.12) such that C' contains at least one atom a_1^p, $1 \leq p \leq w_1$, and for every other disjunct y of C', $y \in \mathcal{N}$. Since $N = |\mathcal{N}|$, there are $\binom{N+w_1}{K-l+1} - \binom{N}{K-l+1}$ such clauses C'. Since when generating T_W from S we do not remove any clauses, the formula (4.13), case 2, follows.

With the same assumption, a clause C belongs to $F_{W,x}$ if and only if C is obtained from a clause C' in S of the form (4.11) such that C' contains at least one atom a_1^p, for some $1 \leq p \leq w_1$, and for every other disjunct y of C', $y \in \mathcal{P}$. Since $P = |\mathcal{P}|$, there are $\binom{P}{u+1} - \binom{P-w_1}{u+1}$ such clauses C', which is also the number of clauses C. Thus the formula (4.14), case 2, holds.

For the formula (4.15), again, a clause C belongs to $G_{W,x}$ if and only if one of the two conditions listed in case 1 is true. Since this time $x \in W$ and $I \models x$, the number of the first type of the clauses becomes $\binom{P-w_1}{u+1}$, while the number of the second type of clauses does not change. Therefore, case 2 of the formula (4.15) holds.

Case 3. The reasoning is similar to that in case 2. This time x occurs in W and $I \not\models x$. Again we assume that $x = a_1$. Since $I \not\models a_1$, $\{a_1^1, \ldots, a_1^{w_1}\} \cap \mathcal{P} = \emptyset$. The definitions of S and $E_{W,x}$ imply that $C \in E_{W,x}$ if and only if C is obtained from a clause C' in S of the form (4.11) such that C' contains at least one atom a_1^p, $1 \leq p \leq w_1$, and for every other disjunct y of C', $y \in \mathcal{P}$. Since $P = |\mathcal{P}|$, there are $\binom{P+w_1}{u+1} - \binom{P}{u+1}$ such clauses C'. Since when generating T_W from S we do not remove any clauses, the formula (4.13), case 3, follows.

With the same assumption, a clause C belongs to $F_{W,x}$ if and only if C is obtained from a clause C' in S of the form (4.12) such that C' contains at least one atom a_1^p, for some $1 \leq p \leq w_1$, and for every other disjunct y of C', $y \in \mathcal{N}$. Since $N = |\mathcal{N}|$, there are $\binom{N}{K-l+1} - \binom{N-w_1}{K-l+1}$ such clauses C', which is also the number of clauses C. Thus the formula (4.14), case 3, holds.

For the formula (4.15), a clause C belongs to $G_{W,x}$ if and only if one of the two conditions listed in case 1 is true. Since this time $x \in W$ and $I \not\models x$, the number of the second type of the clauses becomes $\binom{N-w_1}{K-l+1}$, while the number of the first type of clauses does not change. Therefore, case 3 of the formula (4.15) holds. □

We now use the following example to illustrate how the virtual break- and make-count of an atom are computed.

Example 33. *Let* $C = W_1 \vee W_2 \vee W_3$, *where* $W_1 = 2[a_1, a_2, a_3]2$, $W_2 = 4[2a_2, 1a_3, 4a_4]5$, *and* $W_3 = 3[10a_5, 3a_3, 8a_6]10$.

Let $I = \{a_1, a_3, a_4, a_5\}$ *be the truth assignment. We first note that:*

1. *in* W_1, $K_1 = 3, N_1 = 1, P_1 = 2$
2. *in* W_2, $K_2 = 7, N_2 = 2, P_2 = 5$
3. *in* W_3, $K_3 = 21, N_3 = 8, P_3 = 13$.

Now suppose we flip atom a_2. We recall that $I \not\models a_2$. Based on the formulas of e, f, g, we have the following result:

1. *in W_1, since $a_2 \in W_1$, case 3 applies:*

$$e_1 = \binom{P_1 + w_1^2}{u_1 + 1} - \binom{P_1}{u_1 + 1} = \binom{2+1}{2+1} - \binom{2}{2+1} = 1$$

$$f_1 = \binom{N_1}{K_1 - l_1 + 1} - \binom{N_1 - w_1^2}{K_1 - l_1 + 1} = \binom{1}{3-2+1} - \binom{1-1}{3-2+1} = 0$$

$$g_1 = \binom{P_1}{u_1 + 1} + \binom{N_1 - w_1^2}{K_1 - l_1 + 1} = \binom{2}{2+1} + \binom{1-1}{3-2+1} = 0$$

2. *in W_2, since $a_2 \in W_2$, case 3 applies:*

$$e_2 = \binom{P_2 + w_2^2}{u_2 + 1} - \binom{P_2}{u_2 + 1} = \binom{5+2}{5+1} - \binom{5}{5+1} = 7$$

$$f_1 = \binom{N_2}{K_2 - l_2 + 1} - \binom{N_2 - w_2^2}{K_2 - l_2 + 1} = \binom{2}{7-4+1} - \binom{2-2}{7-4+1} = 0$$

$$g_1 = \binom{P_2}{u_2 + 1} + \binom{N_2 - w_2^2}{K_2 - l_2 + 1} = \binom{5}{5+1} + \binom{2-2}{7-4+1} = 0$$

3. *in W_3, since $a_2 \notin W_3$, case 1 applies:*

$$e_3 = f_3 = 0$$

$$g_3 = \binom{P_3}{u_3 + 1} + \binom{N_3}{K_3 - l_3 + 1} = \binom{13}{10+1} + \binom{8}{21-3+1} = 78$$

Then we can compute the break- and make-count of a_2 using formula (4.9) and (4.10):

$$\textit{break-count}_C(a_2) = \prod_{i=1}^{3}(e_i + g_i) - \prod_{i=1}^{3} g_i = (1+0) \times (7+0) \times (0+78) - 0 \times 0 \times 78 = 546$$

$$\textit{make-count}_C(a_2) = \prod_{i=1}^{3}(f_i + g_i) - \prod_{i=1}^{3} g_i = (0+0) \times (0+0) \times (0+78) - 0 \times 0 \times 78 = 0$$

△

We now use the formulas for break- and make-counts in Figure 4.8 and Figure 4.9. We call the resulting two implementations $wsat(wa)$-skc and $wsat(wa)$-rnp respectively.

Clearly, CNF representations of PB constraints other than the one proposed in this chapter are possible and could be used within a general approach we have developed, as long as one can derive formulas (or procedures) to compute values of e, f and g. In fact, we can push this idea even further. For an arbitrary constraint (not necessarily a pb-constraint), if we can evaluate e, f and g in some translation that converts it into a set of propositional clauses, our general framework yields solvers accepting theories containing such constraints.

Finally, we point out that, in the formulas we have derived, we use values of the form $\binom{n}{k}$, which will exceed the maximum integer that a computer can represent even for relatively small values of n, if k is close to $n/2$. For instances we used in our experiments, even though in some cases overflows occurred quite often (which we replaced with a certain fixed large integer), for the atoms our solvers selected to flip the computation of virtual counts only rarely involved overflows. Still, in our future research we will study how to approximate $\binom{n}{k}$ to avoid overflows. Since we only care about the relative order of the break- and make-counts of atoms, any approximation that maintains this ordering will be appropriate.

4.2.2 Double flip procedure

The second type of instantiations of the algorithm $SLS\text{-}generic$ that we will discuss applies only to PL^{wa} theories of some special syntactic form. Let A be a w-atom of the form $l[a_1 = w_1, \ldots, a_k = w_k]u$. We say that A is **trivial** if $A \equiv \bot$ or $A \equiv \top$; we say that A is **simplifiable** if A logically entails a_i or $\neg a_i$ for some a_i in A.

Example 34. *Let $A = 10[a_1 = 8, a_2 = 3, a_3 = 3, a_4 = 21]20$. Since the upper bound of A is 20 and the weight a_4 is 21, we must set a_4 to false in order to satisfy A. That is, A logically entails $\neg a_4$. Furthermore, since the lower bound of A is 10, we must set atom a_1 to true in order to satisfy A. Therefore, A logically entails a_1.* $\triangle$

Definition 24. *A PL^{wa} theory T is* **simple***, if $T = T^u \cup T^{nu}$, where $T^u \cap T^{nu} = \emptyset$ and*

1. *T^u consists of* **unit** *clauses W_i, where W_i is a c-atom*[2]*, $1 \leq i \leq p$, such that sets of atoms in W_i are pairwise disjoint*

2. *for every i, $1 \leq i \leq p$, W_i is neither trivial nor simplifiable.*

Condition (2) is not particularly restrictive. We can always simplify a PL^{wa} theory that violates only condition (2) and obtain an equivalent **simple** PL^{wa} theory.

Definition 25. *A simple PL^{wa} theory T is* **strictly simple***, if all clauses in T^{nu} are propositional clauses.*

In this section, we consider simple PL^{wa} theories. Let us assume that we design the procedure $Flip(T, I, a)$ so that it has the following property:

(DF) if a truth assignment I is a model of T^u then $I' = Flip(T, I, a)$ is also a model of T^u.

Let us consider a try starting with a truth assignment I that satisfies all clauses in T^u. If our procedure $Flip$ satisfies the property (DF), then all truth assignments that we generate in this try satisfy all clauses in T^u. It follows that the only clauses that can become unsatisfied during the try are the clauses in T^{nu}. Consequently, we can compute the virtual break-count and make-count with respect to T^{nu} only. Furthermore, if T is strictly simple, then all clauses in T^{nu} are propositional. In that case, we only need to consider the CNF theory T^{nu} and count how many clauses in T^{nu} become unsatisfied or satisfied when we perform a flip.

Since all c-atoms in T^u are pairwise disjoint, it is easy to generate random truth assignments that satisfy all these constraints. Thus, it is easy to generate a random starting truth assignment for a try. Moreover, it is also quite straightforward to design a procedure $Flip$ so that it satisfies property (DF). We outline one such procedure now. A detailed pseudo-code description is given in Figure 4.10.

Let us assume that I is a truth assignment that satisfies all clauses in T^u and that we select an atom a as the third argument for the procedure $Flip$. If flipping the value of a does

[2]In general, W_i could be a w-atom. However, it is computationally more expensive to keep W_i satisfied all the time if W_i is a general w-atom, as fixing a broken w-atom often requires more than one flip.

INPUT: T - a simple PL^{wa} theory ($T = T^u \cup T^{nu}$)
I - current truth assignment
a - an atom chosen to flip
OUTPUT: I - updated I after a is flipped
BEGIN
1. **If** a occurs in a clause in T^u **and** flipping a will break it **then**
2. pick an atom b in the clause such that flipping it will fix the clause;
3. $I \leftarrow I^{\bar{b}}$;
4. **End if**
5. $I \leftarrow I^{\bar{a}}$;
6. **return** I;
END

Figure 4.10: Algorithm $Flip(T, I, a)$

not violate any unit clause in T^u, the procedure $Flip(T, I, a)$ returns the truth assignment obtained from I by flipping the value of a. Otherwise, since the c-atoms forming the clauses in T^u are pairwise disjoint, there is exactly one clause in T^u, say W, that becomes unsatisfied when the value of a is flipped. In this case, clearly, $a \in W$.

We proceed now as follows. We find in W an atom, say b, such that flipping both a and b makes W satisfied. Such b exists because W is neither trivial nor simplifiable. Clearly, by performing this **double flip** we maintain the property that all clauses in T^u are still satisfied. Indeed all clauses in T^u other than W are not affected by the flips (these clauses contain neither a nor b).

We implement one instance of $wsat(cc)$-$generic$ using the double flip procedure, which uses the SKC heuristic to pick atoms a and b. We call it $wsat(wa)$-df.

Finally, we want to point out that we could extend the T^u to the case where each unit clause is formed by a w-atom. However, in general, we need to flip a set of atoms to maintain the truth value of a w-atom if flipping a will make it unsatisfied. We will investigate ways of finding such set of atoms in our future work.

Chapter 5

Experimental results

In this chapter, we conduct experimental study on the implementations of the solvers we have proposed so far. To be precise, we implement the following solvers:

1. $pbmodels$: a family of solvers that compute stable models of $lparse$-programs via different PB solvers and PL^{wa} solvers. The PB solvers we use in this chapter are: *satzoo* [45], **pbs** [1], and **wsatoip** [132]. The first two are complete PB solvers while the last one is an incomplete PB solver based on WSAT [123] local search algorithm. We also plug our PL^{wa} solvers showed below into $pbmodels$.

2. $wsat(wa)$: a family of SLS solvers that compute models of PL^{wa}-theories. There are three of them: $wsat(wa)$-skc, $wsat(wa)$-rnp, and $wsat(wa)$-df.

We compare $pbmodels$ to $smodels$, the existing native solver that computes stable models of $lparse$-programs [125]. We test them on instances of six NP-search problems. The results show that $pbmodels$ performs better than $smodels$ on instances of five problems and is comparable to $smodels$ on instances of the last problem.

The architecture of $pbmodels$ allows the use of SLS PB and PL^{wa} solvers as the back-end computing engine. The experimental results also show that $pbmodels$ with SLS solvers outperforms $smodels$ significantly in instances of four problems that have solutions.

With the development of PB solvers [96], we expect that $pbmodels$' performance will be improved as well.

The development of $wsat(wa)$ solvers was motivated by the fact that direct PL^{wa} solvers fit better in the $pbmodels$ architecture than PB solvers. In this chapter, we also show that performance-wise, $wsat(wa)$ performs better than SLS PB solvers on instances of six NP-search problems. In particular, three of these problems require the use of boolean combinations of PB-constraints in their PL^{wa} encodings. They do not have straight-forward

encodings as sets of PB-constraints. Therefore, SLS PB solvers suffer from the overhead caused by the transformation from PL^{wa} theories into PB-theories.

The chapter is organized as follows: in Section 5.1 we describe the setup of our experiments and the comparison measures we focus on; in Section 5.2, we show the experimental results of comparing $pbmodels$ to $lparse$-program solvers; in Section 5.3, we show experimental results of comparing $wsat(wa)$ to SLS PB solvers.

5.1 Experiment setup

We use four identical machines in all of our experiments. Each of these machines is equipped with a Pentium 4 3.2GHz CPU, 1GB memory, running Linux with kernel version 2.6.10. The compiler we use to compile the source codes of solvers is gcc 3.3.4 [1]. For $smodels$ and $cmodels$, we use the default compilation flags provided by the software packages. For our solvers, we compile them with the following two flags: "-Wall -O3".

We set the run-time parameters of each solver we test to its default values except for SLS solvers. For SLS solvers, command-line parameters such as **MAX-TRY**, **MAX-FLIP**, and **p**(the noise ratio) greatly affect their performance. In Section 4.2, we perform a systematic experimental study on the performance of SLS solvers with different values of the parameters, especially with different values of the noise ratio. In testing $pbmodels$ combined with SLS solvers, however, for simplicity, we only use the values of those parameters that give the best performance to the SLS solvers.

The general schema of our experiments is as follows:

1. We choose several benchmark problems from domains such as graph theory, puzzles, and planning. For each benchmark problem, we generate a family of random instances, or otherwise, when there is no easy way to generate random instances, we pick a family of determined instances. The $lparse$ and PL^{wa} encodings of these problem are given in Appendix A.

[1] We could not obtain the source code of $wsat(oip)$. The binary code we use was obtained from the author's website [131].

2. We set a 1000-second run-time limit to each solver we test. There is no limitation, other than the physical one, on the memory usage.

All solvers we test report the amount of time they spend to *solve* an instance. By **solving an instance**, we mean either the solver finds a model (or stable model) or it decides that no models (or stable models) exist. However, the way in which they measure the timing information is not consistent among these solvers. Some of them report the CPU time, but others report the wall-time. For a fair comparison, we use the GNU time program (version 1.7) to gather the timing information of all solvers. We report or perform statistical analysis on the "user time" reported by the GNU time program, which is the CPU time spent by the solvers.

For comparison, we report the following statistics of all solvers tested for each benchmark problem. We first compare the solvers instance by instance. A solver **wins** in that instance if the amount of time it uses to solve the instance is the minimum one (no other solver uses less time). We report, for each benchmark problem, in how many instances a solver wins. Then we aggregate the timing information of a solver in the family of random instances by means of the **run-time distribution** (or RTD for short) instead of the simple statistical measures such as average or standard deviation.

The reason is that, experiments show that the hardness of instances generated randomly with fixed parameters varies significantly. Therefore, the run time of a solver solving an instance, which can be viewed as a random variable, varies significantly as well. In other words, the probability distribution on the run time has high variance. In this case, simple statistics such as average run time does not provide enough information about the performance of different solvers. Researchers have studied the similar problem in evaluating stochastic local search solvers. Hoos and Stützle [71] propose that we should use the run-time distribution as a more accurate and realistic measure of the performance behavior of local search solvers. They focus on estimating and characterizing the run-time distribution over a single instance, because the run-time of a local search solver is a random variable even on a single instance.

We estimate the run-time distribution of a solver over a family of randomly generated

instances as follows. We run the solver on each instance and record the amount of time it takes to solve the instance. If a solver does not terminate when the 1000-second time limit is reached, we say the solver fails to solve this instance. Then we estimate the probability for a solver S to solve an instance in the family F within time $0 \leq t \leq 1000$ by the ratio M/N, where M is the number of instances that are solved by S within time t and N is the total number of instances in that family. More precisely, $Pr_S^F(T \leq t) = M/N$.

5.2 Comparing *pbmodels* with *lparse*-program solvers

In this section, we present our experimental results concerning the performance of $pbmodels$.

In the experiments we use instances of the following benchmark problems: **traveling salesperson**, **weighted n-queens**, **weighted Latin square**, **magic square**, **vertex cover**, and **towers of Hanoi**. The $lparse$-programs we used for the first four problems involve general w-atoms. The $lparse$-programs for the last two problems only use c-atoms.

The experiments compare $pbmodels$ to $smodels$ on several sets of benchmark instances. In most cases $pbmodels$ outperforms its competitors. We also test $cmodels$ in the experiment because $pbmodels$ is based on the idea of $cmodels$ [7]. However, $cmodels$ does not perform well in most of the instances: either $cmodels$ times out or it gives a segmentation fault. The only benchmark problem in which $cmodels$ can solve some instances is the **vertex cover** problem. Even there, $cmodels$ is outperformed by both $smodels$ and $pbmodels$ with most of the PB solvers as the back-end search engines.

We now give the description of each benchmark problem we use in our experiments.

1. **Traveling salesperson problem (tsp).** An instance consists of a weighted complete graph of n vertices and a bound w. Edge weights and w are non-negative integers. A solution to an instance is a Hamiltonian cycle whose weight (the sum of the weights of its edges in the cycle) is less than or equal to w.

   We randomly generate 50 weighted complete graphs with 20 vertices. The weight of each edge is chosen uniformly from the range $[1..19]$. By setting w to 100 we obtain an "easy" set, denoted by **TSP-e**, of 50 instances (the bound is high enough for every

instance in the set to have a solution). We also create another set, denoted by **TSP-h**, of 50 instances from the same collection of graphs by setting w to 62. This set is "harder" as the constraint on the weight of the cycle is stronger. Consequently, we find solutions only for about half of these instances. For the remaining ones we either determine that they do not have solutions or none of the solvers we test terminated with success within the time limit of 1000 seconds.

2. **Weighted n-queens problem (wnq).** An instance consists of a weighted $n \times n$ chess board and a bound w. All weights and the bound are non-negative integers. A solution to an instance is a placement of n queens on the chess board so that

   - no queen attacks another, and
   - the weight of the placement (the sum of the weights of the squares with queens) is not greater than w.

   We randomly generate 50 weighted chess boards of the size 20×20 (each chess board is associated with a set of 20×20 weights $w_{i,j}$, $1 \leq i, j \leq 20$, which is chosen uniformly from the range $[1..19]$). We then create two sets of instances, easy and hard, by setting the bound w to 70 and 50, respectively. We denote the two sets of instances **wnq-e** and **wnq-h** respectively.

3. **Weighted Latin square problem ($wlsq$).** An instance consists of an $n \times n$ array W and a bound w. All entries in W and the bound w are non-negative integers. A solution to an instance is an $n \times n$ array L with all entries from $\{1, \ldots, n\}$ and such that

   - each element in $\{1, \ldots, n\}$ occurs exactly once in each row and each column in the array L, and
   - $\sum_{i=1}^{n} \sum_{j=1}^{n} L[i, j] \times W[i, j] \leq w$.

   We randomly generate 50 arrays W of size 10×10 with values uniformly randomly

chosen from the range $[1..10]$. Again we create two families of instances, easy (**wlsq-e**) and hard (*wlsq-h*), by setting w to 280 and 225, respectively.

4. **Magic square problem** (msq). An instance consists of a positive integer n. The goal is to construct an $n \times n$ array using each integer $1, \ldots n^2$ as an entry in the array exactly once in such a way that entries in each row, each column and in both main diagonals sum up to $n(n^2+1)/2$. For the experiments we use the magic square problem for $n = 4, 5$ and 6.

5. **Vertex cover problem** ($vcov$). An instance consists of graph of n vertices and m edges, and a non-negative integer k — a bound. A solution to the instance is a subset of vertices of the graph with no more than k vertices and such that at least one end vertex of every edge of the graph is in the subset.

   We choose the **vertex cover** problem for experiments for two reasons. First, the most direct encoding of the problem uses cardinality constraints only. While the way $cmodels$ complies away weight atoms is not very effective, it generally handles cardinality atoms quite well. Thus, we could perform meaningful tests of $cmodels$ on programs encoding instances of that problem. Second, we expect the programs encoding instances of the vertex cover problem to be challenging for $cmodels$. The difficulty is due to the fact that the upper bound in the cardinality atom in these programs may be quite large and require superlinear representations as CNF theories.

   We randomly generate 50 graphs each with 80 vertices and 400 edges. For each graph, we set k to be a smallest integer such that a vertex cover with that many elements still exists.

6. **Towers of Hanoi problem** (toh). This is a generalization of the original problem. We consider the case with six disks. An instance consists of an initial configuration of disks that satisfies the constraint of the problem (larger disk must not be on top of a smaller one). These configurations were selected so that they were $31, 36, 41, 63$ steps away from the goal configuration, respectively. We also consider a standard

Benchmark	*smodels*	*pbmodels-satzoo*	*pbmodels-pbs*
magic square (4×4)	1.36	1.70	2.41
magic square (5×5)	> 1000	28.13	0.31
magic square (6×6)	> 1000	75.58	> 1000
towers of Hanoi $(d = 6, t = 31)$	16.19	18.47	1.44
towers of Hanoi $(d = 6, t = 36)$	32.21	31.72	1.54
towers of Hanoi $(d = 6, t = 41)$	296.32	49.90	3.12
towers of Hanoi $(d = 6, t = 63)$	> 1000	> 1000	3.67
towers of Hanoi $(d = 7, t = 127)$	> 1000	> 1000	22.83

Table 5.1: $pbmodels$ v.s. $smodels$: Magic square and towers of Hanoi problems

version of the problem with seven disks, in which the initial configuration is 127 steps away from the goal.

In order to test $pbmodels$ and $smodels$, we encode the constraints of each benchmark problem along with the instances we generate as $lparse$ logic programs. Those families of $lparse$-programs form inputs to $pbmodels$ and $smodels$.

In the tests, we use $pbmodels$ with the following four PB and PL^{wa} solvers: $satzoo$ [45], pbs [1], $wsat(oip)$ [132], and $wsat(wa)$. As we have mentioned, we use default values of the command-line parameters of each solver. For $wsat(wa)$, we only report the result of $wsat(wa)$-rnp implementation as $wsat(wa)$-rnp performs better than $wsat(wa)$-skc in these benchmark problems. For both $wsat(wa)$-rnp and $wsat(oip)$, we set their command-line parameters to the values that give them the best performance. That is, we set **MAX-TRY** to 10, **MAX-FLIP** to 2,000,000, and **p** to 0.1.

In the figures and tables below we report results from our experiments.

We first show the results for the **magic square** and **towers of Hanoi** problems in Table 5.1. For each solver and each instance we report the corresponding running time in seconds.

Both $pbmodels$-$satzoo$ and $pbmodels$-pbs perform better than $smodels$ on programs obtained by encoding instances of both problems. We observe that $pbmodels$-pbs performs exceptionally well in the tower of Hanoi problem. It is the only solver that can compute a plan for 7 disks, which requires 127 steps. Despite the fact that the tower of Hanoi problem involves only cardinality constraints, $cmodels$ times out on every instance we tested. Local-

	# of SAT instances	*# of UNSAT instances*	*# of UNKNOWN instances*
TSP-e	50	0	0
TSP-h	31	1	18
wnq-e	49	0	1
wnq-h	29	0	21
wlsq-e	45	4	1
wlsq-h	8	41	1
vtxcov	50	0	0

Table 5.2: *pbmodels* v.s. *smodels*: Summary of Instances

	smodels	*pbmodels-satzoo*	*pbmodels-pbs*
TSP-e	45/17	50/30	18/3
TSP-h	7/3	16/14	0/0
wnq-e	11/5	26/23	0/0
wnq-h	2/2	0/0	0/0
wlsq-e	21/1	49/29	46/19
wlsq-h	0/0	47/26	47/23
vtxcov	50/40	50/1	47/3
sum over all	136/68	238/123	158/48

Table 5.3: *pbmodels* v.s. *smodels*: Summary on all instances

search solvers were unable to solve any of the instances in the two problems and so are not included in the table.

For the remaining four problems, we use 50-element families of instances, which we generate randomly in the way discussed above. We study the performance of complete solvers (*smodels*, *pbmodels-satzoo* and *pbmodels-pbs*) on all instances. We then include local-search solvers (*pbmodels-wsatcc* and *pbmodels-wsatoip*) in the comparisons but restrict attention only to instances that were determined to be satisfiable (as local-search solvers are, by their design, unable to decide unsatisfiability). In Table 5.2, for each family we list how many of its instances are satisfiable, unsatisfiable, and for how many of the instances none of the solvers we test was able to decide satisfiability.

In Table 5.3, for each of the seven families of instances and for each **complete** solver, we report two values s/w, where s is the number of instances solved by the solver and w is the number of times it won in the test (the fastest among the three).

The results in Table 5.3 show that overall *pbmodels-satzoo* solved more instances than *pbmodels-pbs*, followed by *smodels*. When we look at the number of times a solver was

	smodels	*pbmodels-satzoo*	*pbmodels-pbs*	*pbmodels-wsat(wa)-rnp*	*pbmodels-wsatoip*
TSP-e	45/3	50/5	18/2	32/7	47/34
TSP-h	7/0	16/2	0/0	19/6	28/22
wnq-e	11/0	26/0	0/0	49/45	49/4
wnq-h	2/0	0/0	0/0	29/15	29/14
wlsq-e	21/0	45/0	44/0	45/33	45/14
wlsq-h	0/0	7/0	8/0	7/1	8/7
vtxcov	50/0	50/0	47/0	50/36	50/15
sum over all	136/3	194/7	117/2	231/143	256/110

Table 5.4: $pbmodels$ v.s. $smodels$: Summary on SAT instances

the fastest one, $pbmodels$-$satzoo$ was a clear winner overall, followed by $smodels$ and then by $pbmodels$-pbs. Looking at the seven families of tests individually, we see that $pbmodels$-$satzoo$ performs better than the other two solvers on five of the families. On the other two $smodels$ was the best performer (although, it is a clear winner only on the vertex-cover benchmark; all solvers were essentially ineffective on the *wnq-h*).

We also study the performance of $pbmodels$ combined with local-search solvers $wsat(wa)$ and $wsat(oip)$ [132]. For this study, we consider only those instances in the seven families that we knew were satisfiable. Table 5.4 presents results for all solvers we study (including the complete ones). As before, each entry provides a pair of numbers s/w, where s is the number of solved instances and w is the number of times the solver performs better than its competitors.

The results show superior performance of $pbmodels$ combined with local-search solvers. They solve more instances than complete solvers (including $smodels$). In addition, they are significantly faster, winning much more frequently than complete solvers do (complete solvers were faster only on 12 instances, while local-search solvers were faster on 253 instances).

Our results demonstrate that $pbmodels$ with solvers of pseudo-boolean constraints outperforms $smodels$ on several types of search problems involving pseudo-boolean (weight) constraints.

Next, we discuss our observations based on the RTDs of these solvers in the experiment. These RTDs further confirm our observations. All the RTD plots are presented in Appendix

B.

According to those figures:

1. We observe that the run-time distribution of $pbmodels$ with $satzoo$ is consistently above that of $smodels$ in five out of seven families of instances. That implies $pbmodels$ with $satzoo$ performs better than $smodels$ in those five families. In the **weighted latin square** hard instance family, $smodels$ only outperforms $pbmodels$ with $satzoo$ by less than 0.05% when $t \geq 750$. Therefore, $smodels$ does not have a significant win in this family of instances.

2. The difference between the run-time distribution of $smodels$ and that of $pbmodels$ with $satzoo$ increases when the family of instances changes from the easy one to the hard one in the **TSP** and **weighted latin square** problems. It shows that $smodels$ does not scale well in those two problems as instances become hard.

3. $Smodels$ is outperformed by $pbmodels$ with PB solvers, especially with local search PB solvers. We also have experiments that show $pbmodels$ with local search PB solvers scales better than $smodels$ when the size of the instances increases.

4. The **vertex cover** family is the only one in which $cmodels$ could solve some instances. From the run-time distribution of $cmodels$, we observe that $cmodels$ is consistently worse than PB with $wsat(oip)$, $wsat(wa)$ and $satzoo$ and worse than $smodels$. This observation supports our conjecture that the compilation of cardinality constraints with large bounds (such as the one in the vertex cover problem) has an adverse effect the performance of solvers that require that step.

5.3 Comparing $wsat(wa)$ with PB SAT solvers

In this section, we further test the implementations of our algorithms on PL^{wa}-theories encoding instances of six search problems.

Solvers for PL^{wa}-theories are important both because $pbmodels$ uses PL^{wa} as the target logic for computing the completion and the loop formulas and because logic PL^{wa} itself is

an effective declarative programming formalism that subsumes logic PB. In other words, we can use PL^{wa}-theories directly to model search problems, without the need to start from logic programs and go through the completion and the loop formula construction. We have illustrated how to model a search problem directly in logic PL^{wa} in Section 3.5, Chapter 3. Consequently, $wsat(wa)$ is also an independent computational tool by which we can solve search problems directly encoded as PL^{wa}-theories. Therefore, it is important to see how well $wsat(wa)$ performs.

We compare the performance of our solvers to that of $wsat(oip)$ [132], the only known SLS PB solver in the literature. We show that our solvers designed directly for PL^{wa}-theories perform better than $wsat(oip)$, which requires non-trivial transformation from PL^{wa}-theories into PB-theories. This result implies that the development of direct PL^{wa} solvers is important and, potentially, will improve the performance of $pbmodels$ as well.

We now discuss our experiments in detail. First, let us take a look at the six search problems we use in this section:

1. **Vertex-cover problem (vcov).** The problem is the same as $vcov$ problem defined in Section 5.2. There, in order to test complete solvers, we did not use large graphs. In this section, we are testing SLS solvers. Thus we generate 50 random graphs of 2000 vertices and 4000 edges. For each graph, we then select a relatively small integer as the bound for the size of the vertex cover in such a way that the problem still had a solution.

2. **Traveling salesperson problem (tsp).** The problem is the same as the one defined in Section 5.2. In this experiment, we generate 50 random weighted complete graphs of 40 vertices. The weight of each edge is uniformly chosen from the range $[1..39]$. Then for each instance, we then select a relatively small integer as the TSP bound in such a way that the problem still had a solution.

3. **Bounded spanning tree problem (bst).** Let $G = (V, E)$ be an undirected graph and w a positive integer. Each edge $\{x, y\} \in E$ has a weight $w_{x,y} \geq 0$. The goal is to find a spanning tree T of G such that for each vertex $x \in V$, the sum of the weights

of all edges in T incident to x is at most w.

We generate 50 random graphs of 30 vertices and 240 edges. The weight of each edge is uniformly chosen from the range $[1..29]$. We set w to 15 so that all instances are satisfiable.

4. **Weighted k-coloring problem (wcol)** This is a variant of the graph k-coloring problem. In this variant, we assign weights to edges in the graph. That is, each edge $\{u, v\}$ has a non-negative weight denoted by $w_{u,v}$. We then require that 1) for each color the sum of weights of edges whose two ends receive the same color is at most p; and 2) there exists at least one color such that the sum of weights of edges whose two ends receive that color at the same time is at least q.

5. **Weighted dominating set problem (wdm).** The problem is defined earlier in Chapter 1.

   We generate 50 random graphs of 500 vertices and 2000 edges. The weight of each edge is generated uniformly from $[1..19]$. The value of w is set to 40 and k to 330 so that all instances we generate are satisfiable.

6. **Weighted n-queens problem with distance constraint (dwnq).** This problem is a variant of the weighted n-queens problem we have introduced in Section 5.2. Every square (i, j) on an $n \times n$ chess board has a weight $w_{i,j} \geq 0$. Given two integers $w \geq 0$ and $d \geq 0$, the goal is to find an arrangement of n queens on the board so that 1) no two queens attack each other; 2) the sum of weights of the squares with queens does not exceed w; and 3) for each queen Q, there is at least one queen Q' in a neighboring row or column such that the Manhattan distance between Q and Q' is greater than or equal to d.

   We generate 50 random weighted 20×20 chess boards, where the weights are uniformly chosen from range $[1..19]$. The value of w is set to 80 and d to 10. All instances we generate are satisfiable.

We use the first two problems in testing *pbmodels* as well. However, the instances

we generate in this section are different from those generated in the previous section. The reason is that those instances generated in the previous section are too easy for SLS solvers. The other four problems are new. The **bst** problem is interesting in itself. The other three problems involve boolean combinations of PB-constraints that do not have a straight-forward encoding in PB.

We present the PL^{wa}-theory encoding an instance of the problem **wdm** earlier in the thesis. Encodings for other problems can be found in Appendix A. We note that:

1. Theories for the **vcov** problem consist only of strict pb-constraints and are accepted directly by our programs and $wsat(oip)$. Thus, they can be processed without any modifications by our programs as well as by solvers of strict pb-constraints.

2. Theories for the **tsp** problem and the **bst** problem contain formulas which are not strict pb-constraints. However, these formulas have simple representations as one or two strict pb-constraints and do not require the help of new atoms. In experiments, we use original encodings with our algorithms and transformed encodings with $wsat(oip)$.

3. Theories encoding instances of the last three problems consist of non-unary PL^{wa}-clauses. To avoid a blow-up in the size of representation, when expressing these clauses in terms of sets of pb-constraints, we **need** to introduce new atoms. As before, we use original encodings with our algorithms and transformed encodings with $wsat(oip)$.

Since the choice of the noise ratio p often has strong effect on the performance of $wsat(wa)$-rnp, we test all methods with 9 different noise ratios $0.1, 0.2, \ldots, 0.9$. For comparisons, we use results obtained with the best value of p for each method.

For each instance, we ran each solver in one try, with the maximum number of flips set so that to guarantee the unsuccessful try does not end prior to the 1000-second limit. We set other parameters of each solver to their default settings.

We first present a summary of all experiments in Table 5.5. It shows two values in the form s/w. Value s denotes the number of instances a solver solved in a family of instances.

	$wsat(wa)$-skc	$wsat(wa)$-rnp	$wsat(wa)$-df	$wsat(oip)$
vcov	$30/0$	$\mathbf{48/35}$	$47/10$	$42/4$
tsp	$1/0$	$\mathbf{50/48}$	NA	$50/2$
bst	$50/10$	$\mathbf{50/41}$	NA	$50/0$
wcol	$50/0$	$50/0$	$\mathbf{50/49}$	$1/1$
wdm	$49/25$	$\mathbf{50/26}$	$49/0$	$4/0$
dwnq	$\mathbf{50/38}$	$46/11$	NA	$2/0$

Table 5.5: $wsat(wa)$ v.s. $wsat(oip)$: summary on all instances

Value w denotes the number of instances that it solves with the shortest amount of time among all solvers we tested.

These results demonstrate the superiority of our methods over $wsat(oip)$ on the instances we use in experiments. Of the two methods we proposed, $wsat(wa)$-rnp performs better in three out of four problems, with $wsat(wa)$-skc being significantly better for the remaining one. We emphasize that our algorithms perform better than $wsat(oip)$ even for problems that were encoded directly as sets of strict pb-constraints or required only small and simple modifications (problems **vcov** and **bst**). There is only one exception: for the problem **vcov** $wsat(oip)$ outperforms $wsat(wa)$-skc (but is outperformed by $wsat(wa)$-rnp).

We now present RTD graphs for the problems **bst** and **wdm** problem. RTDs for other problems can be found in Appendix C. Figure 5.1 shows that $wsat(wa)$-rnp performs the best.

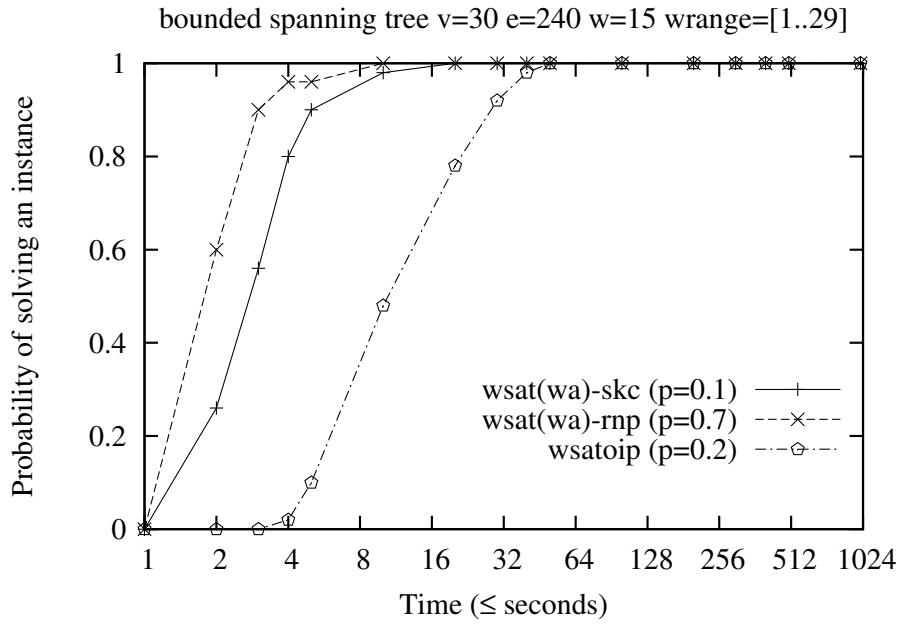

Figure 5.1: RTDs on the bst problem

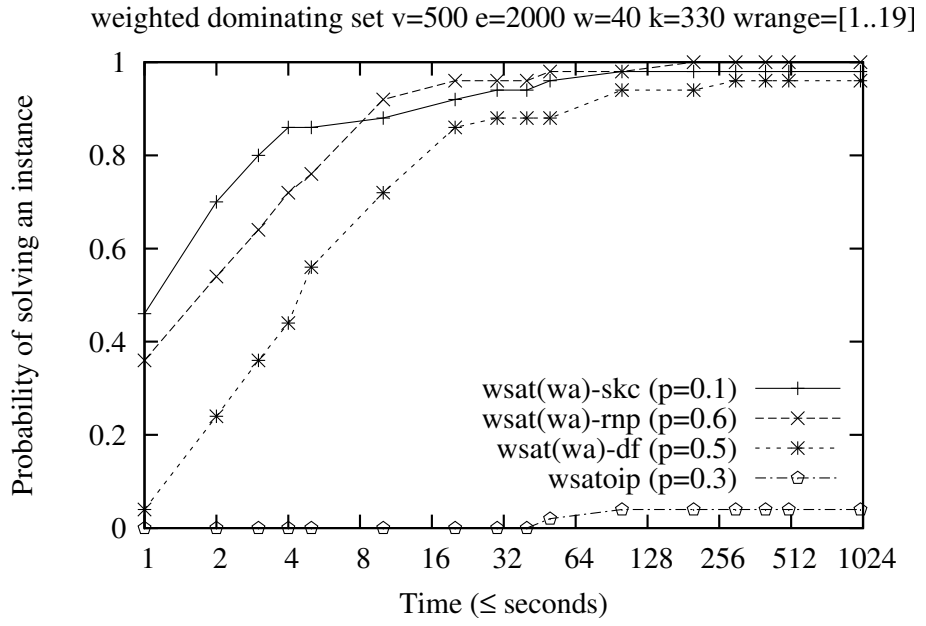

Figure 5.2: RTDs on the wdm problem

Figure 5.2 shows that $wsat(oip)$ is not effective. It also shows that $wsat(wa)$-skc has a higher probability of solving easy instances (instances that can be solved in up to about 8 seconds). Then $wsat(wa)$-rnp catches up and the performance of the two algorithms is very similar, with $wsat(wa)$-rnp being slightly better (in fact, it is the only algorithm that solves all instances in the family).

Finally, we discuss the **robustness** of a local search solver with respect to the noise ratio. The noise ratio is an important parameter to all $wsat$-like solvers. Experiments show that the behavior of a $wsat$-like solver, measured by its run-time distribution, on some input theories may vary significantly if the noise ratio is changed slighted. Furthermore, to solve different input theories, a solver may need different noise ratios to achieve its best behavior. Therefore, setting the noise ratio is not a trivial task.

A local search solver is **robust on an instance** P **with respect to the noise ration** if its run-time distribution on P does not depend on the value of the noise ratio. A robust solver makes the task of choosing the best noise ratio easier. We note that researchers [69] have studied the problem of learning the best noise ratio by solvers themselves. We do not address this issue in the dissertation.

To study the robustness of variants of $wsat(wa)$ and $wsat(oip)$, we gather the run-time distributions of the solvers using nine different noise ratios on a family of test instances. We construct a 3D plot using these data, where x-axis measures the noise ratio, y-axis measures the running time, and z-axis measures the probability of solving an instance in the family. A solver is robust on the family of instances if the plot shows a collection of similar curves along the x-axis. We need to point out that if a solver is robust it does not necessarily mean the solver performs well on the instance. For example, a solver can be completely ineffective (not being able to solve the instance) no matter which noise ratio is used. Then its run-timing distribution will not change at all when the noise ratio changes.

Figure 5.3 shows the plot for $wsat(wa)$-skc on the family of instances from the problem **vcov**.

We observe that the best run-time distribution of $wsat(wa)$-skc changes dramatically when the noise ratio changes from 0.1 to 0.2 and larger. With the noise ratio being 0.1,

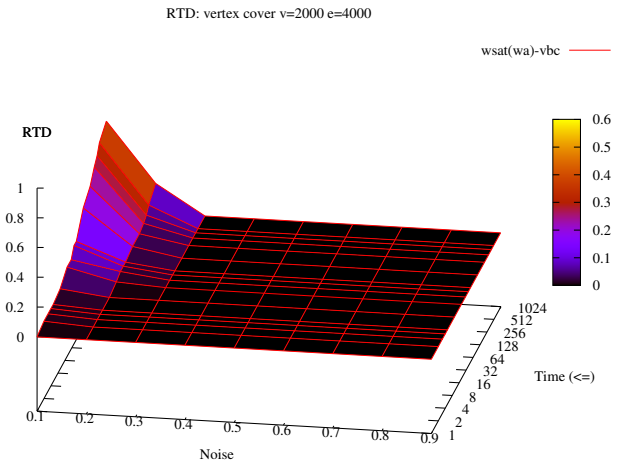

Figure 5.3: $vcov$: $wsat(wa)$-skc

$wsat(wa)$-skc is able to achieve a peak at 0.6 on the z-axis. When the noise ratio goes up to 0.2, the peak drops to 0.2. When the noise ratio is greater than 0.2, the curve of the run-time distribution becomes a line on the x-y surface.

Figure 5.4, 5.5, 5.6 show the plots for $wsat(wa)$-rnp, $wsat(wa)$-df, and $wsat(oip)$ on the same family of instances respectively.

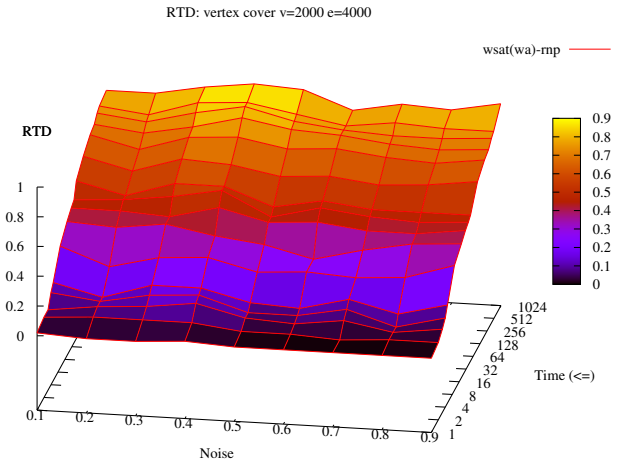

Figure 5.4: $vcov$: $wsat(wa)$-rnp

It is clear that $wsat(wa)$-rnp is much more robust than $wsat(wa)$-skc on this family

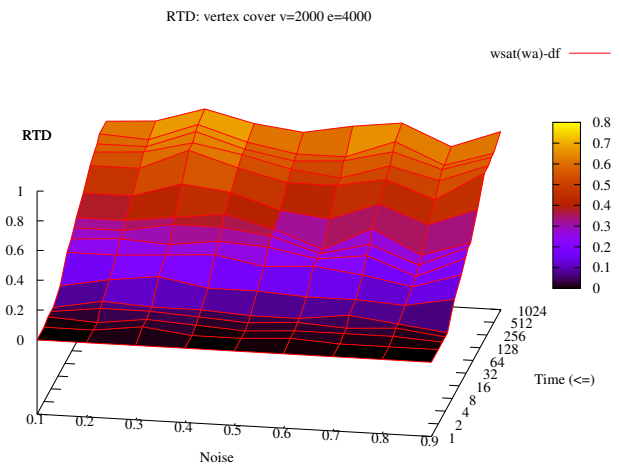

Figure 5.5: $vcov$: $wsat(wa)$-df

of instance. Actually, if we compare Figure 5.4 to Figure 5.5 and 5.6, we observe that $wsat(wa)$-rnp is the most robust solver among all four solvers in this family of instances. Solver $wsat(wa)$-df is the second most robust solver. $wsat(oip)$ is slightly more robust than $wsat(wa)$-skc. However, it becomes completely ineffective when the noise ratio is greater than 0.3.

The set of data comparing the robustness of the solvers on every family of instances is given in Appendix D. The conclusion on this comparison is that $wsat(wa)$-skc is the most robust solver on these tests, with two exceptions: problem **wrcol** and problem **dwnq**. In problem **dwnq**, the only robust solver is $wsat(oip)$, which is completely ineffective. We note that $wsat(wa)$-df is robust whenever it is applicable.

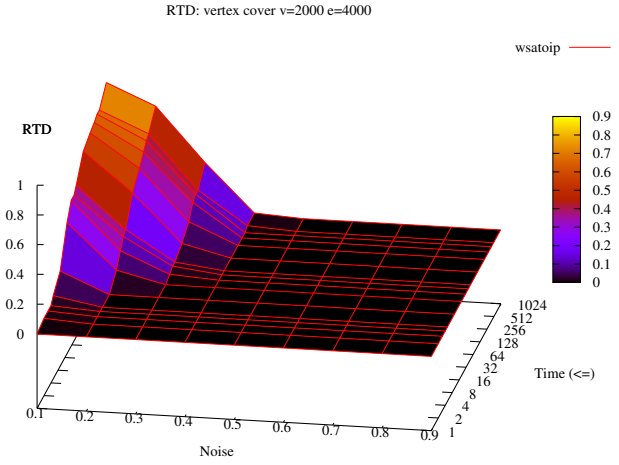

Figure 5.6: $vcov$: $wsat(oip)$

Chapter 6

Conclusions

We addressed the problem of solving hard search problems. We adopted a declarative programming approach. In declarative programming paradigm, we write programs to capture specifications of problems rather than the control algorithms. In particular, we focused on logic programming with stable model semantics. In this formalism, we represent constraints of problems as a collection of logic program rules, called a logic program, such that stable models of the logic program are precisely the solutions to the problem. We write the logic program in a first-order language, where we can use predicates and domain variables. Then we ground the first-order logic program into a propositional one. Finally we compute the stable models of the ground logic program.

In this thesis, we focused on ground logic programs and developed theories concerned with properties of ground logic programs. Researchers have studied normal logic programs for more than two decades, developing the definition of the stable model semantics and properties of normal logic programs. In early 2000's, researchers proposed high-level constructs to the language to facilitate the modeling of constraints on sets. One notable such construct is the weight constraint (or pseudo-boolean constraint), by which we can capture constraints of the weight of a set in a concise way. We call logic programs with such extensions the $lparse$ programs. $Lparse$ programs are the main objects we investigated in this thesis.

The goal of this thesis was to develop theories and algorithms to compute stable models of $lparse$ programs, by which we can solve search problems. Therefore, we first investigated properties of $lparse$ programs. Since researchers proposed $lparse$ programs, most of the research on $lparse$ programs focuses on extending the stable model semantics to this setting. Beyond the definition of stable models of $lparse$ programs, we have not seen much work on their properties.

In the thesis, we considered an even more general type of logic program than $lparse$ programs. We considered logic programs built of abstract monotone or convex constraints. We call such programs mac programs. In particular, $lparse$ programs, with the restriction that only non-negative integers and positive literals occur in a weight constraint, form a subclass of mac programs. We applied an algebraic approach under this setting to define the stable model semantics and to investigate properties of such logic programs. We showed that concepts, techniques and results from normal logic programming generalize to this abstract setting of programs with monotone and convex constraints. Our work differs from related research in the literature because we allow constraints to appear in the heads of program rules. This difference is significant as the one-step provability operator for mac programs becomes non-deterministic.

We studied in properties of mac programs that are related to computing stable models of mac programs. We extended results such as program equivalence, tightness of logic programs, Fages Lemma, program completion, and loop formulas from normal logic programming to this abstract setting. The results we obtained for mac programs specialize to *new* results about $lparse$ programs. While characterizations of strong equivalence of $lparse$ programs were obtained by Turner [129], the characterization of uniform equivalence of $lparse$ programs implied by Theorem 14 is new.

Specializations to the case of $lparse$ of Theorems 16, 19, 22 and 23, concerning Fages Lemma, the completion of a program and loop formulas, are also new.

Given these results we developed an implementation of a new software for computing stable models of $lparse$ programs. The idea is to follow the approach developed in *assat* [81], and use the results we mentioned above to reduce the problem to that of finding models of theories that are boolean combinations of pseudo-boolean constraints, for which several fast solvers exist [38, 82, 87]. We can also convert theories that are boolean combinations of pseudo-boolean constraints into pseudo-boolean SAT instances, for which many effective PB SAT solvers exist [96].

The new software we developed differs from its ancestors *assat* and *cmodels* since it does not compile away weight atoms from logic programs. Therefore, in many cases, our

software performs better than both the native stable model solver *smodels* and compilation-to-SAT stable model solvers *assat* and *cmodels*. Moreover, as the area of PB SAT receives more and more attention and new faster PB solvers are designed, the scope of the effectiveness of $pbmodels$ will continue expanding because our software can take any PB SAT solver as the back-end engine in computing stable models.

As a byproduct of our research on $lparse$ programs, we proposed the extension to propositional logic called logic PL^{wa}, which allows boolean combinations of pseudo-boolean constraints. This logic is of its own interest because of the use of pseudo-boolean constraints. Therefore, we also conducted another line of research to develop solvers for this new logic in the thesis. In particular, we extended the stochastic local search algorithms proposed for SAT to this new logic.

That is, we designed a family of extensible SLS algorithms for PL^{wa} theories. The key idea behind our algorithms is to view a PL^{wa} theory T as a concise representation of a certain propositional CNF theory $cl(T)$ logically equivalent to T, and to show that key parameters needed by SLS solvers developed for CNF theories can be computed on the basis of T, without the need to build $cl(T)$ explicitly. Our experiments demonstrate that our methods are superior to those relying on explicit representations of PL^{wa} clauses as sets of PB constraints (a.k.a. PB SAT instances) and resorting to off-the-shelf local-search solvers for PB constraints such as $wsat(oip)$.

We performed extensive experimental study at the end of this dissertation. The results show that our research has yield two types of solver that are significantly better than existing solvers in the literature.

There are still open questions in this research that may constitute future investigations:

1. extending our results to disjunctive logic programming with pseudo-boolean constraints.

2. defining new type of program equivalence. Current definition of strong or uniform equivalence is too restrictive. Therefore, it is hard to use these two types of equivalence in practical applications such as program optimization or building libraries.

We want to weaken the definition of program equivalence, yet still can replace one program module by another without changing the meaning of the overall program.

3. improving $pbmodels$ algorithm. Current algorithm treats PB solvers as blackboxes. That is, when a supported but not stable model is found, the algorithm computes its loop formula, adds it to the theories, and calls the PB solver. The PB solver in this case will start its search process all from scratch. In other words, the PB solver loses all the information it gained during previous search processes. We want to modify $pbmodels$ algorithm so that it treats PB solvers as grayboxes, which can pause the search process, wait for $pbmodels$ to add loop formulas, and resume the search process from where it was stopped.

4. finding ways to deal with overflow problem in computing virtual counts in $wsat(wa)$. As we have mentioned, the virtual counts can cause overflows fairly easily. Since we only care about the ordering of atoms defined by the virtual counts, we want to develop effective and fast approximations of virtual counts so that: 1) few or no overflow will occur when we use the approximations; and 2) they maintain the correct ordering by the accurate virtual counts.

5. extending $wsat(wa)$ to deal with more types of constraint. As we pointed out in Chapter 4, the idea of virtual counts can be pushed even further. Given an arbitrary constraint (not necessarily a PB constraint), as long as we can find an equivalent propositional logic representation that has a regular structure, we can apply our virtual count computation on this constraint. This extension will enable $wsat(wa)$ to even more general settings.

Appendix A $Lparse$ and Logic PL^{wa} encodings of the benchmark problems

We show encodings of the benchmark problems we used in our experiments, the RTD plots and the comparison of solvers on the robustness in this chapter.

A.1 Vertex cover problem

We build the theory $T_{vc}(G, k)$ of atoms in_i, $1 \leq i \leq n$, (intended meaning of in_i: vertex i is in a vertex cover) and define it to consist of the following clauses:

VCC1: $\{in_1, \ldots, in_n\}k$

VCC2: $in_p \vee in_r$

for every edge $(p, r) \in E$

Clause (VCC1) (a single one) guarantees that at most k vertices are chosen to a vertex cover. Clauses (VCC2) enforce the main vertex cover constraint. That is, for each edge (p, r), either p is in the subset or r is in the subset.

The logic program that represents **vertex cover** problem is similar.

VCR1: $\{in_1, \ldots, in_n\}k \leftarrow$

VCR2: $\leftarrow \mathbf{not}(in_p), \mathbf{not}(in_r)$

for every edge $(p, r) \in E$

Rule (VCR1) is used to guess a subset of vertices of size less than or equal to k. Then rules (VCR2) ensure the subset we guess from rule (VCR1) is indeed a vertex cover.

A.2 Traveling salesperson problem

To encode the problem as a PL^{wa}-theory, say $T_{tsp}(G, k)$, we use atoms $p_{i,v}$ and $e_{u,v}$, where $1 \leq i \leq |V|$ and $u, v \in V$. The atoms $p_{i,v}$ define a permutation of V specifying a Hamiltonian cycle. Intuitively, an atom $p_{i,v}$ captures the statement that v is in the position i in

the permutation. Auxiliary atoms $e_{u,v}$ are meant to represent the edges $\{u, v\}$ that form a selected Hamiltonian cycle. The theory $T_{tsp}(G, k)$ consists of the following clauses:

TC1: $1[p_{i,v} \colon v \in V]1$

for every i, $1 \leq i \leq |V|$

TC2: $1[p_{i,v} \colon 1 \leq i \leq |V|]1$

for every $v \in V$

TC3: $\neg p_{i,u} \vee \neg p_{i+1,v} \vee e_{u,v}$

for every $u, v \in V$ and for every i, $1 \leq i \leq n$ (when incrementing indices by one, we assume here that $n + 1 = 1$)

TC4: $\neg e_{u,v} \vee \neg p_{i,u} \vee p_{i+1,v}$

for every $u, v \in V$ and for every i, $1 \leq i \leq n$ (as before, we assume that $n + 1 = 1$)

TC5: $[e_{u,v} = w_{u,v} : u, v \in V]k$

The first two sets (TC1) and (TC2) of clauses together ensure the permutation constraints. The second pair of sets (TC3) and (TC4) of clauses define the edges $e_{u,v}$ that form the Hamiltonian cycle determined by a permutation $p_{i,v}$. Finally, clause (TC5) enforces the bound on the length of a cycle.

The logic program $P_{tsp}(G, k)$ is given as follows:

TR1: $1[p_{i,v} \colon v \in V]1$

for every i, $1 \leq i \leq |V|$

TR2: $1[p_{i,v} \colon 1 \leq i \leq |V|]1$

for every $v \in V$

TR3: $e_{u,v} \leftarrow p_{i,u}, p_{i+1,v}$

for every $u, v \in V$ and for every i, $1 \leq i \leq n$ (when incrementing indices by one, we assume here that $n + 1 = 1$)

TR4: $\leftarrow k + 1[e_{u,v} = w_{u,v} : u, v \in V]$

A.3 Bounded spanning tree problem

To build a spanning tree, we pick exactly $n-1$ edges from the graph. These edges form a spanning tree if and only if we can find a permutation of vertices such that the following property holds:

(ST) for any vertex at position $i > 1$ in the permutation, there must have exactly one vertex at position $1 \leq j < i$ such that there is a tree edge between them.

Indeed, if these edges form a spanning tree T, then we perform a breadth-first search. The order in which vertices are visited forms a permutation since the spanning tree connects all vertices in G. Since vertices in G are connected in T, every vertex at position $i > 1$ of the permutation has an ancestor. Assume the permutation does not satisfy the property (ST). Then there exists a position $i > 1$ such that the vertex u at this position has edges to more than one vertex at position $1 \leq j < i$. Then one can construct a cycle in this case. It contradicts the assumption that T is a spanning tree. Therefore the permutation satisfies the property (ST).

For the other direction, let T be the edges we choose, and let P be the permutation of vertices that satisfies property (ST). In order to prove the edges in T form a spanning tree, We need to show that the edges do not form cycles. Assume it is not the case, then there exists a cycle $(v_1, \ldots, v_k)$ such that v_i and v_{i+1} are connected by an edge in T, for $i = 1, \ldots, k$ and $k+1 = 1$. Since vertices $v_1, \ldots, v_k$ appear in the permutation P, without loss of generality, we assume v_1 is the vertex in the cycle that occurs the earliest in the permutation. Since v_2 occurs after v_1 in the permutation and there is an edge in T between v_2 and v_1, then v_3 must occur after v_2 because the permutation satisfies property (ST). With the same reasoning, we infer that v_k occurs the last in the permutation. However, since there is an edge between v_1 and v_k, v_k has two vertices: v_1 and v_{k-1} that occur before it and have edges in T to v_k. It is a contradiction. Therefore, the edges in T form a spanning tree.

To build PL^{wa}-theory, $T_{bst}(G, k)$, encoding the bounded spanning tree problem, we use atoms $sv_{i,x}$ and $te_{x,y}$ ($i = 1, \ldots, |V|$, $x, y \in V$). Atoms $sv_{i,x}$ establish a permutation of vertices in G. The intended meaning of $te_{x,y}$ is that edge (x, y) is chosen into the spanning

tree. In order to model the property (ST) concisely, we view edges represented by $te_{x,y}$ as directed edges from x to y. Thus the property (ST) can be modified to the following one:

(ST'-1) a vertex at position $i > 1$ in the permutation has exactly one incoming tree edge from a vertex at position $1 \leq j < i$;

(ST'-2) if there is a tree edge from x to y, then x appears before y in the permutation.

Once the spanning tree is built, the bound constraint is easy to capture. We include in $T_{bst}(G, w)$ the following clauses:

SC1: $(n-1)\{te_{x,y} \colon \{x,y\} \in E\}(n-1)$

SC2: $\neg te_{x,y}$

for every non-edge pair $\{x, y\}$

SC3: $1\{sv_{i,x} : x \in V\}1$

for every $i = 1, \ldots, n$

SC4: $1\{sv_{i,x} : i = 1, \ldots n\}1$

for every $x \in V$

SC5: $\neg sv_{i,y} \vee C$

where $C = 1\{te_{z,y} \colon \forall z, \{z,y\} \in E\}1$

for every $i > 1$ and $y \in V$

SC6: $\neg sv_{i,x} \vee \neg sv_{j,y} \vee \neg te_{x,y}$

for every $1 \leq j < i \leq n$ and $x, y \in V$

SC7: $\{te_{x,y} = w_{x,y} \colon \forall x, z, \{x,y\} \in E\}w$

for all $y \in V$

It is clear that clauses (SC1) and (SC2) enforce that exactly $n-1$ edges are chosen into the spanning tree. Clauses (SC3) and (SC4) generate a permutation of all vertices in G. Clauses (SC5) capture the property (ST'-1) and clauses (SC6) capture the property (ST'-2). Finally clauses (SC7) ensure the weight bound constraint on each vertex.

Now we give an *lparse*-program that represents the same problem. We use the same set of atoms that occur in PL^{wa}-theory $T_{bst}(G,k)$. The *lparse*-program $P_{bst}(G,k)$ contains the following rules:

SR1: $(n-1)\{te_{x,y}\colon \{x,y\} \in E\}(n-1) \leftarrow$

SR2: $1\{sv_{i,x} : x \in V\}1 \leftarrow$

for every $i = 1, \ldots, n$

SR3: $1\{sv_{i,x} : i = 1, \ldots n\}1 \leftarrow$

for every $x \in V$

SR4: $C \leftarrow sv_{i,y}$

where $C = 1\{te_{z,y}\colon \forall z, \{z,y\} \in E\}1$

for every $i > 1$ and $y \in V$

SR5: $\leftarrow sv_{i,x}, sv_{j,y}, te_{x,y}$

for every $1 \leq j < i \leq n$ and $x, y \in V$

SR6: $\leftarrow w + 1\{te_{x,y} = w_{x,y}\colon \forall x, z, \{x,y\} \in E\}$

for all $y \in V$

The *lparse*-program $P_{bst}(G,k)$ is almost the same as the PL^{wa} theory $T_{bst}(G,k)$. The only exception is that clauses (SC2) do not have a correspondence in program $P_{bst}(G,k)$. The reason is that the only ways to derive atoms $te_{x,y}$ are through rules (SR1) and (SR4). They have restricted that only those $te_{x,y}$'s such that $\{x,y\}$ is an edge in G could ever be derived (or assigned value $\mathtt{t}$ in any stable model).

A.4 Weighted *k*-coloring problem

We use the following PL^{wa}-theory T_{pcol} to represent the weighted relaxed k-coloring problem. We introduce a set of propositional atoms: $c_{i,c}$ and $e_{i,j,c}$, where $1 \leq i, j \leq n$ and $1 \leq c \leq k$. The intended meaning of $c_{i,c}$ is that vertex i has color c, and the intended meaning of $e_{i,j,c}$ is the two ends of edge (i,j) receive the same color c.

CC1: $1\{c_{i,1}, \ldots, c_{i,k}\}1$
for every i, $1 \leq i \leq n$.

CC2: $\neg c_{i,c} \vee \neg c_{j,c} \vee e_{i,j,c}$
for every edge $\{i, j\}$ in the graph and every color $1 \leq c \leq k$

CC3: $\neg e_{i,j,c} \vee c_{i,c}$
for every edge $\{i, j\}$ and every color $1 \leq c \leq k$

CC4: $\neg e_{i,j,c} \vee c_{j,c}$
for every edge $\{i, j\}$ and every color $1 \leq c \leq k$

CC5: $\neg e_{i,j,c}$
for every non-edge pair $\{i, j\}$ and every color $1 \leq c \leq k$

CC6: $[e_{i,j,c} = w_{i,j} : \{i, j\} \in E]p$
for every color $1 \leq c \leq k$

CC7: $W_1 \vee \ldots \vee W_k$
where $W_x = q[e_{i,j,c_x} = w_{i,j} : \{i, j\} \in E]$
for $x = 1, \ldots, k$

Clauses (CC1) ensure that every vertex obtains **exactly one** color. Clauses (CC2) to (CC5) enforce that $e_{i,j,c}$ is $\mathbf{t}$ if and only if $\{i, j\}$ is an edge whose two ends receive the same color c. Clauses (CC6) ensure the constraint 1) of the problem. Finally, clause (CC7) ensures the constraint 2) of the problem.

Now let us take a look at the logic program encoding for the same variant.

CR1: $1\{c_{i,1}, \ldots, c_{i,k}\}1 \leftarrow$

for every i, $1 \leq i \leq n$

CR2: $e_{i,j,c} \leftarrow c_{i,c}, c_{j,c}$

for every edge $\{i, j\}$ in the graph and every color $1 \leq c \leq k$

CR3: $\leftarrow p + 1[e_{i,j,c} = w_{i,j} : (i, j) \in E]$

for every color $1 \leq c \leq k$

CR4: $\leftarrow W_1, \ldots, W_k$

where $W_x = [e_{i,j,c_x} = w_{i,j} : \{i, j\} \in E]q - 1$

for $x = 1, \ldots, k$

Rules (CR1) ensure that every vertex obtains **exactly one** color. Rules (CR2) compute the edge coloring. Rules (CR3) and (CR4) ensure two constraints of the problem.

A.5 W-Dominating set problem

To construct a theory $T_{dm}(G, k, w)$ encoding the w-dominating-set problem we use atoms of the form in_i, $i \in V$ (this time, in_i stands for: **vertex i is in a dominating set**). Theory $T_{dm}(G, k, w)$ consists of the following clauses:

DC1: $in_i \vee W_1 \vee W_2$

for every $i \in V$, where

$W_1 = w[in_y = w_{i,y} \colon (i, y) \in E]$, and

$W_2 = w[in_z = w_{z,i} \colon (z, i) \in E]$

DC2: $\{in_i \colon i \in V\}k$

Clauses (DC1) enforce the defining constraint for the w-dominating set problem. Clause (DC2) guarantees that a selected subset has at most k vertices.

The logic program $P_{dm}(G, k, w)$ that represents the w-dominating set problem is given below:

DR1: $\leftarrow \mathbf{not}(in_i), W_1, W_2$

for every $i \in V$, where

$W_1 = [in_y = w_{i,y} \colon (i, y) \in E]w - 1$, and

$W_2 = [in_z = w_{z,i} \colon (z, i) \in E]w - 1$

DR2: $\{in_i \colon i \in V\}k$

A.6 Weighted n-queens problem

In the PL^{wa}-theory $T_{wnq}(n, k)$, encoding this problem, we use propositional atoms $q_{i,j}$, $1 \leq i, j \leq n$, to represent occupied blocks: intuitively, $q_{i,j}$ is true if and only if the square (i, j) has a queen. The theory $T_{wnq}(n, k)$ consists of the following clauses:

QC1: $1\{q_{i,1}, \ldots, q_{i,n}\}1$

for every i, $1 \leq i \leq n$

QC2: $1\{q_{1,j}, \ldots, q_{n,j}\}1$

for every j, $1 \leq j \leq n$

QC3: $\neg q_{i,j} \vee \neg q_{k,l}$

for every i, j, k, l such that $1 \leq i < k \leq n$, $1 \leq j, l \leq n$ and $abs(i-k) = abs(j-l)$

QC4: $[q_{i,j} = w_{i,j} : 1 \leq i, j \leq n]k$

Clauses (QC1) ensure that no row contains two or more queens. Similarly clauses (QC2) ensure that no column contains two or more queens. Clauses (QC3) ensure that no diagonal contains two or more queens. The last clause (QC4) enforces the upper bound constraint on the sum of weights of occupied squares.

The following logic program $P_{wnq}(n, k)$ represents the same problem.

QR1: $1\{q_{i,1}, \ldots, q_{i,n}\}1 \leftarrow$

for every i, $1 \leq i \leq n$

QR2: $1\{q_{1,j}, \ldots, q_{n,j}\}1 \leftarrow$

for every j, $1 \leq j \leq n$

QR3: $\leftarrow q_{i,j} \wedge q_{k,l}$

for every i, j, k, l such that $1 \leq i < k \leq n$, $1 \leq j, l \leq n$ and $abs(i-k) = abs(j-l)$

QR4: $\leftarrow k + 1[q_{i,j} = w_{i,j} : 1 \leq i, j \leq n]$

A.7 Weighted n-queens problem with distance constraint

Now we define a logic PL^{wa}-theory $T_{dwnq}(n, k, d)$ that represents this problem. In addition to the set of atoms $q_{i,j}$, we use atoms $mdu_{x,y,z}$, $mdd_{x,y,z}$, $mdl_{x,y,z}$, and $mdr_{x,y,z}$ to denote the Manhattan distance from queen in position (x, y) to its neighbors in row $x - 1$, row $x + 1$, column $y - 1$, column $y + 1$ respectively.

DQC1: $1\{q_{i,1}, \ldots, q_{i,n}\}1$

for every i, $1 \leq i \leq n$

DQC2: $1\{q_{1,j}, \ldots, q_{n,j}\}1$

for every j, $1 \leq j \leq n$

DQC3: $\neg q_{i,j} \vee \neg q_{k,l}$

for every i, j, k, l such that $1 \leq i < k \leq n$, $1 \leq j, l \leq n$ and $abs(i-k) = abs(j-l)$

DQC4: $[q_{i,j} = w_{i,j} : 1 \leq i, j \leq n]k$

DQC5: $\neg q_{x,y} \vee \neg q_{x-1,w} \vee mdu_{x,y,z}$

for every $1 < x \leq n$, $1 \leq y, w \leq n$ and $z = |w - y| + 1$

DQC6: $\neg mdu_{x,y,z} \vee q_{x,y}$

for every $1 \leq x, y, z \leq n$

DQC7: $\neg mdu_{x,y,z} \vee q_{u,v} \vee q_{u,w}$

for every $1 \leq x, y, z \leq n$ and $u = x - 1 \geq 0$, $v = y + z \leq n$, $1 \leq y - z$

DQC8: $\neg q_{x,y} \vee \neg q_{x+1,w} \vee mdd_{x,y,z}$

for every $1 \leq x < n$, $1 \leq y, w \leq n$ and $z = |w - y| + 1$

DQC9: $\neg mdd_{x,y,z} \vee q_{x,y}$

for every $1 \leq x, y, z \leq n$

DQC10: $\neg mdd_{x,y,z} \vee q_{u,v} \vee q_{u,w}$

for every $1 \leq x, y, z \leq n$ and $u = x + 1 \leq n$, $v = y + z \leq n$, $1 \leq y - z$

DQC11: $\neg q_{x,y} \vee \neg q_{w,y-1} \vee mdl_{x,y,z}$

for every $1 < y \leq n$, $1 \leq x, w \leq n$ and $z = |w - x| + 1$

DQC12: $\neg mdl_{x,y,z} \vee q_{x,y}$

for every $1 \leq x, y, z \leq n$

DQC13: $\neg mdl_{x,y,z} \vee q_{v,u} \vee q_{w,u}$

for every $1 \leq x, y, z \leq n$ and $u = y - 1 \geq 0$, $v = x + z \leq n$, $1 \leq x - z$

DQC14: $\neg q_{x,y} \vee \neg q_{w,y+1} \vee mdr_{x,y,z}$

for every $1 \leq y < n$, $1 \leq x, w \leq n$ and $z = |w - x| + 1$

DQC15: $\neg mdr_{x,y,z} \vee q_{x,y}$

for every $1 \leq x, y, z \leq n$

DQC16: $\neg mdr_{x,y,z} \vee q_{u,v} \vee q_{u,w}$

for every $1 \leq x, y, z \leq n$ and $u = y + 1 \leq n$, $v = x + z \leq n$, $1 \leq x - z$

DQC17: $\neg q_{x,y} \vee W_1 \vee W_2 \vee W_3 \vee W_4$

where $W_1 = d[mdu_{x,y,z} = z : 1 \leq z \leq n]$

$W_2 = d[mdd_{x,y,z} = z : 1 \leq z \leq n]$

$W_3 = d[mdl_{x,y,z} = z : 1 \leq z \leq n]$ and

$W_4 = d[mdr_{x,y,z} = z : 1 \leq z \leq n]$

for every $1 \leq x, y \leq n$

Clauses (DQC1) to (DQC4) are from the original **weighted n-queens** theory. Clauses (DQC5) to (DQC7) define atoms $wdu_{x,y,z}$. That is, $wdu_{x,y,z}$ is $\mathbf{t}$ if and only if $q_{x,y}$ is $\mathbf{t}$ and $q_{x-1,w}$ is $\mathbf{t}$ such that $z = |y - w| + 1$. Similarly clauses (DQC8) to (DQC10), (DQC11) to (DQC13), and (DQC14) to (DQC16) define atoms $wdd_{x,y,z}$, $wdl_{x,y,z}$, and $wdr_{x,y,z}$ respectively. The last set of clauses (DQC17) ensure that, for each queen, at least one of its neighbors has distance at least d from it.

Now we define logic program $P_{dwnq}(n, k, d)$ that represents the same problem:

DQR1: $1\{q_{i,1}, \ldots, q_{i,n}\}1$

for every i, $1 \leq i \leq n$

DQR2: $1\{q_{1,j}, \ldots, q_{n,j}\}1$

for every j, $1 \leq j \leq n$

DQR3: $\leftarrow q_{i,j}, q_{k,l}$

for every i, j, k, l such that $1 \leq i < k \leq n$, $1 \leq j, l \leq n$ and $abs(i-k) = abs(j-l)$

DQR4: $\leftarrow k + 1[q_{i,j} = w_{i,j} : 1 \leq i, j \leq n]$

DQR5: $mdu_{x,y,z} \leftarrow q_{x,y}, q_{x-1,w}$

for every $1 < x \leq n$, $1 \leq y, w \leq n$ and $z = |w - y| + 1$

DQR6: $mdd_{x,y,z} \leftarrow q_{x,y}, q_{x+1,w}$

for every $1 \leq x < n$, $1 \leq y, w \leq n$ and $z = |w - y| + 1$

DQR7: $mdl_{x,y,z} \leftarrow q_{x,y}, q_{w,y-1}$

for every $1 < y \leq n$, $1 \leq x, w \leq n$ and $z = |w - x| + 1$

DQR8: $mdr_{x,y,z} \leftarrow q_{x,y}, q_{w,y+1}$

for every $1 \leq y < n$, $1 \leq x, w \leq n$ and $z = |w - x| + 1$

DQR9: $\leftarrow q_{x,y}, W_1, W_2, W_3, W_4$

where $W_1 = [mdu_{x,y,z} = z : 1 \leq z \leq n]d - 1$

$W_2 = [mdd_{x,y,z} = z : 1 \leq z \leq n]d - 1$

$W_3 = [mdl_{x,y,z} = z : 1 \leq z \leq n]d - 1$ and

$W_4 = [mdr_{x,y,z} = z : 1 \leq z \leq n]d - 1$

for every $1 \leq x, y \leq n$

Appendix B RTDs: *pbmodels* v.s. *smodels*

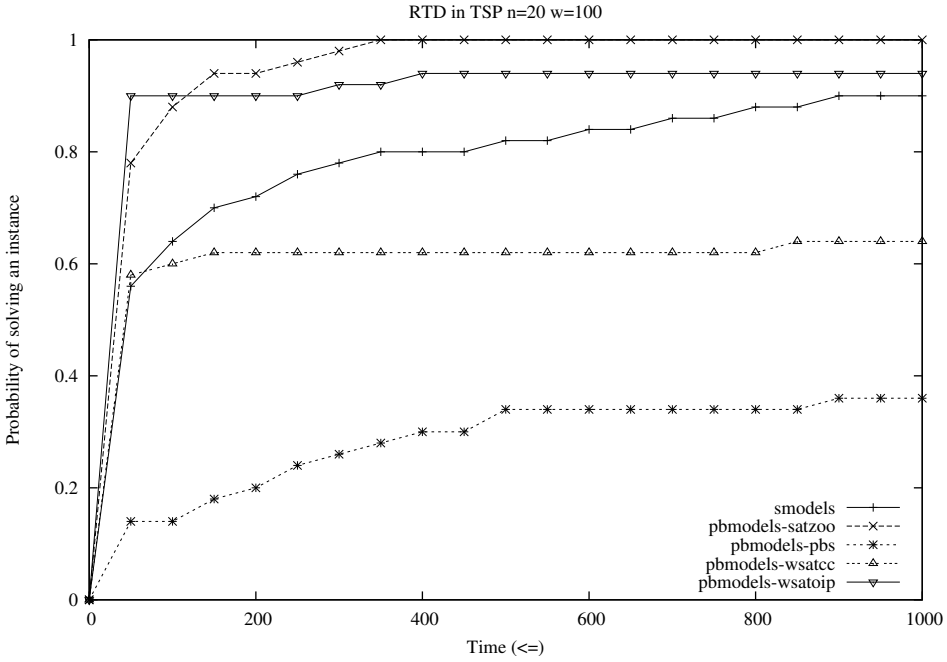

Figure B.1: *pbmodels* v.s. *lparse*: **tsp-e**

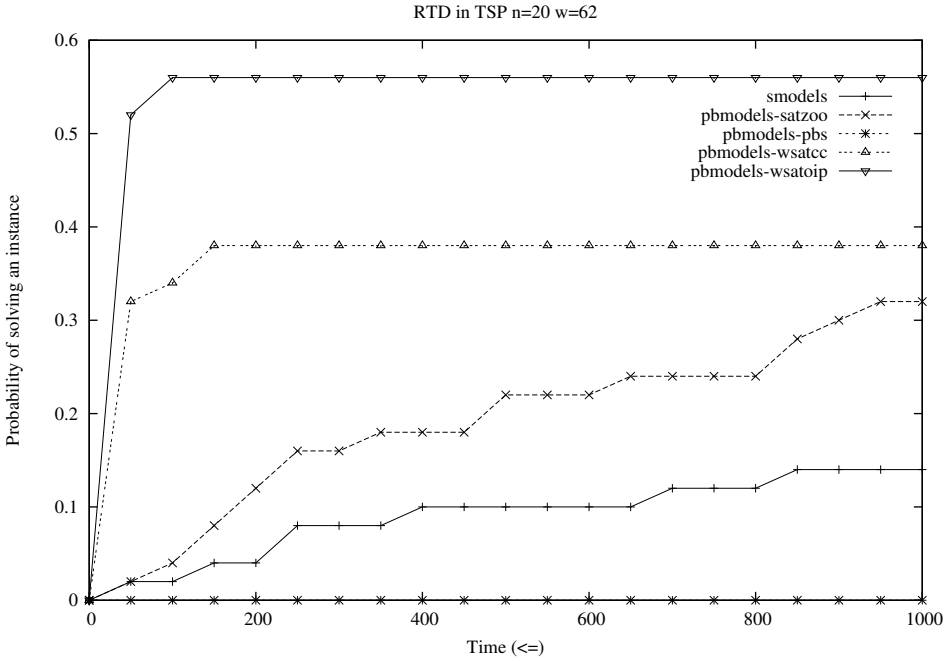

Figure B.2: *pbmodels* v.s. *lparse*: **tsp-h**

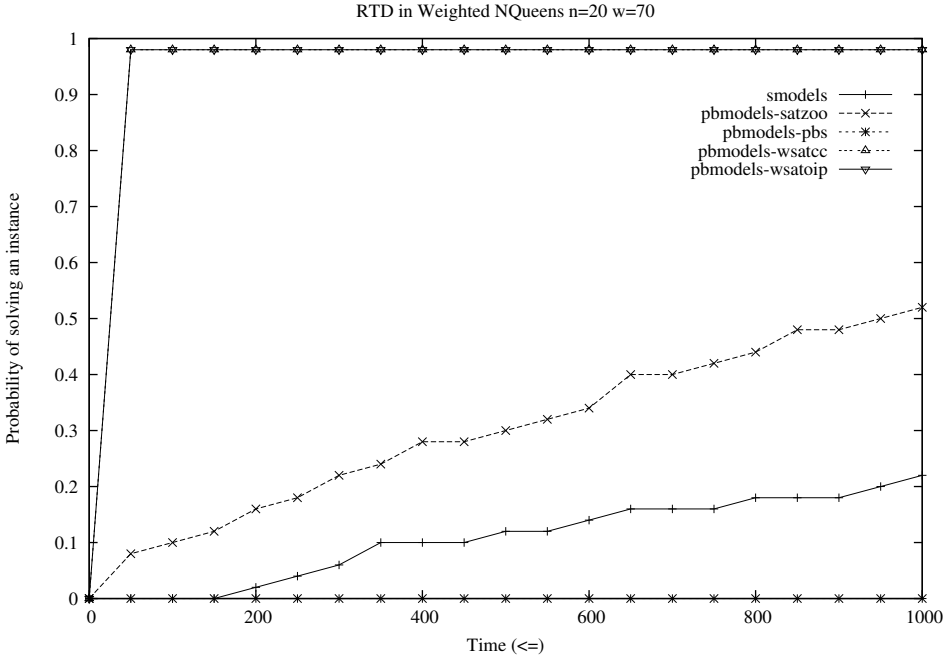

Figure B.3: *pbmodels* v.s. *lparse*: **wnq-e**

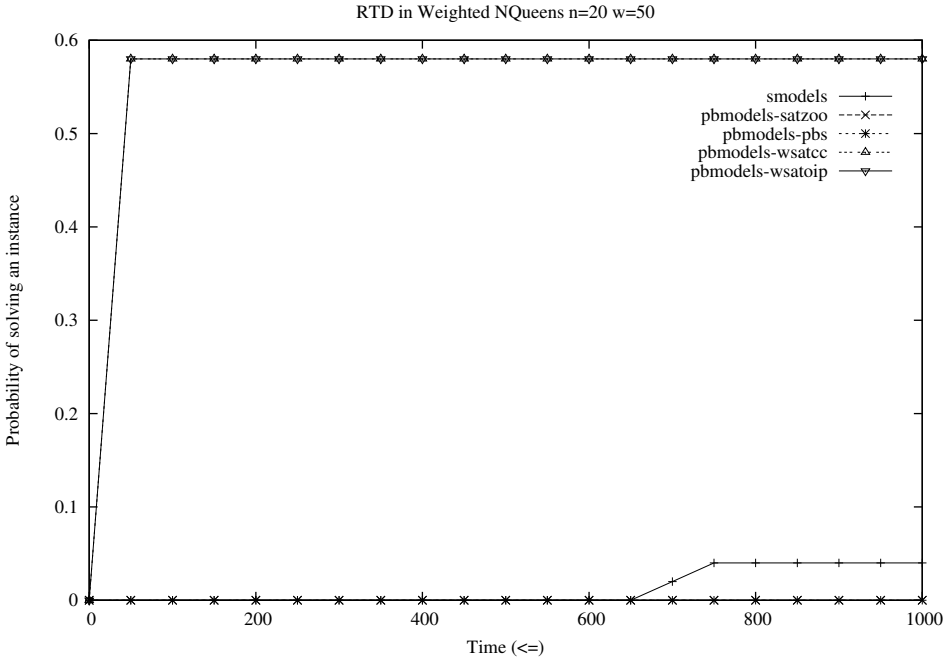

Figure B.4: *pbmodels* v.s. *lparse*: **wnq-h**

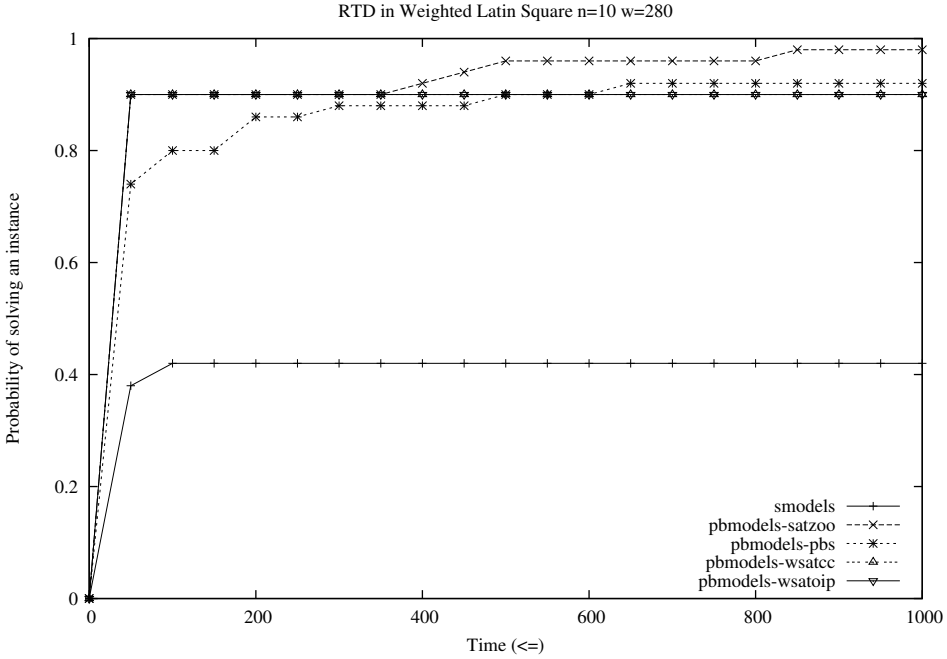

Figure B.5: *pbmodels* v.s. *lparse*: **wls-e**

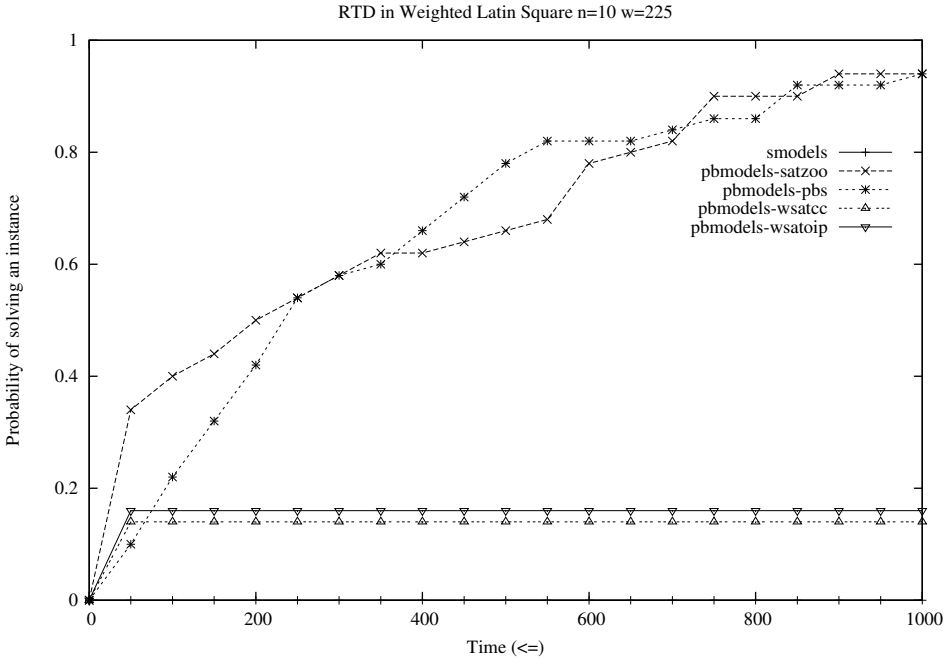

Figure B.6: *pbmodels* v.s. *lparse*: **wls-h**

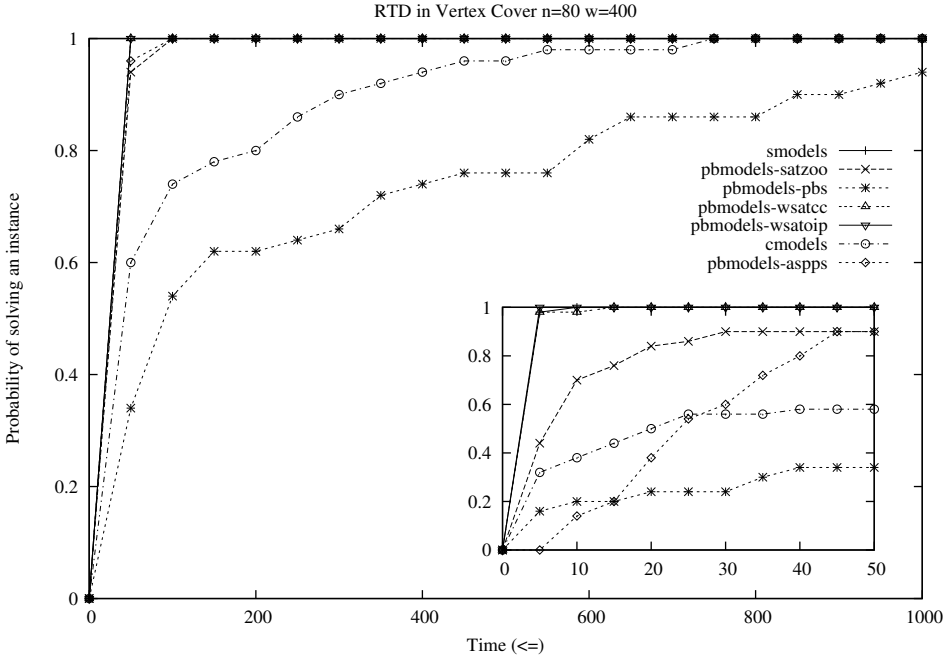

Figure B.7: *pbmodels* v.s. *lparse*: **vtxcov**

Appendix C RTDs: $wsat(wa)$ v.s. $wsat(oip)$

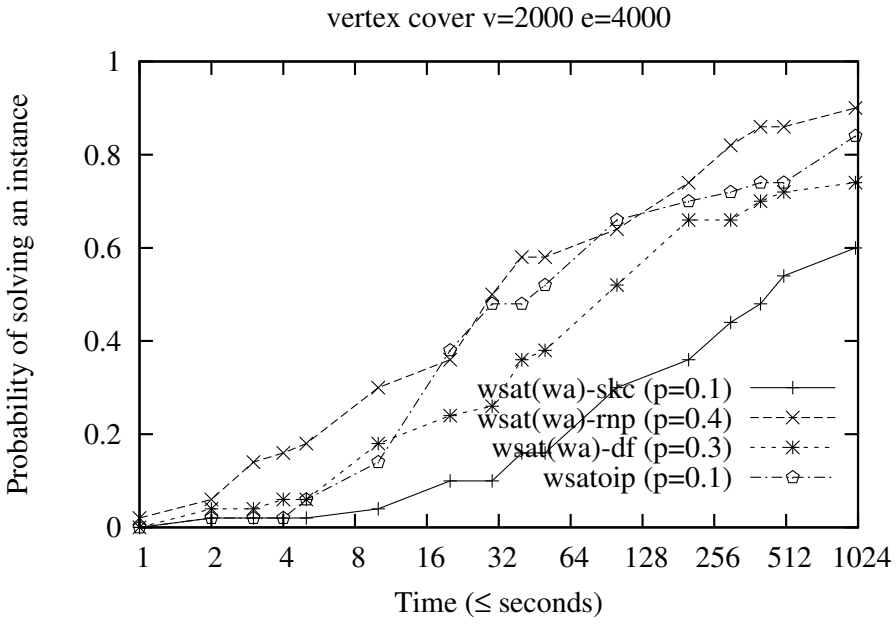

Figure C.1: $wsat(wa)$ v.s. $wsat(oip)$: $vcov$

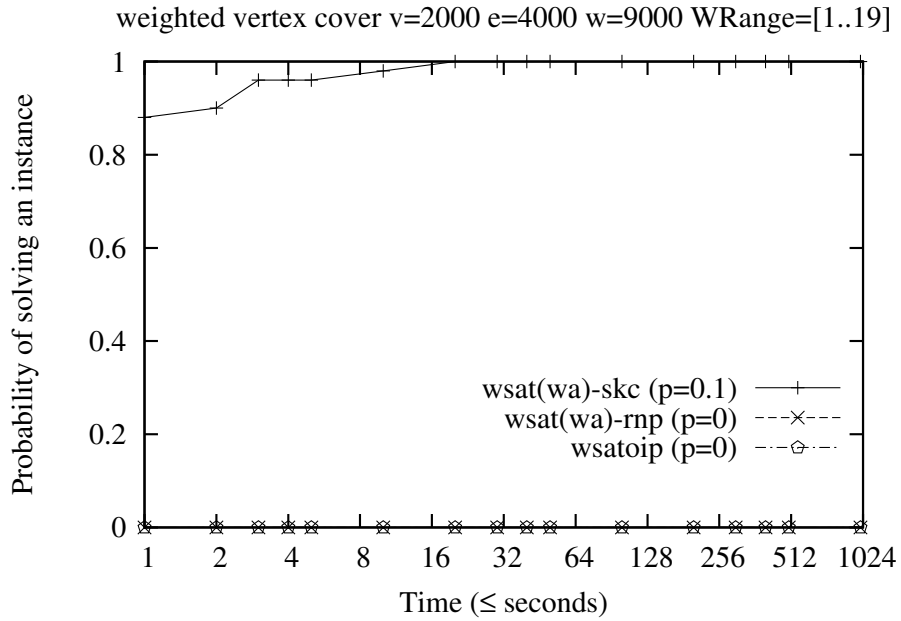

Figure C.2: $wsat(wa)$ v.s. $wsat(oip)$: $wvcov$

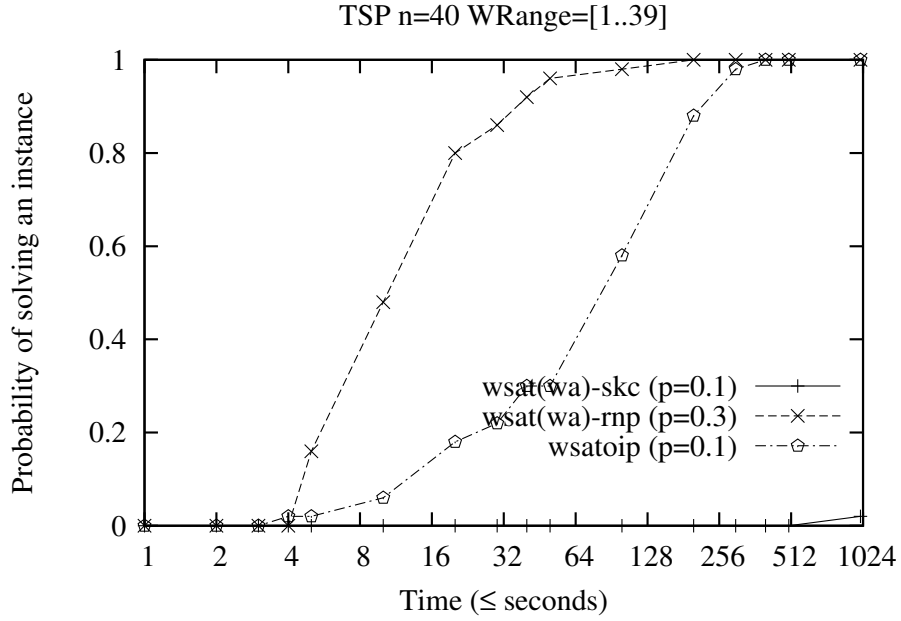

Figure C.3: $wsat(wa)$ v.s. $wsat(oip)$: tsp

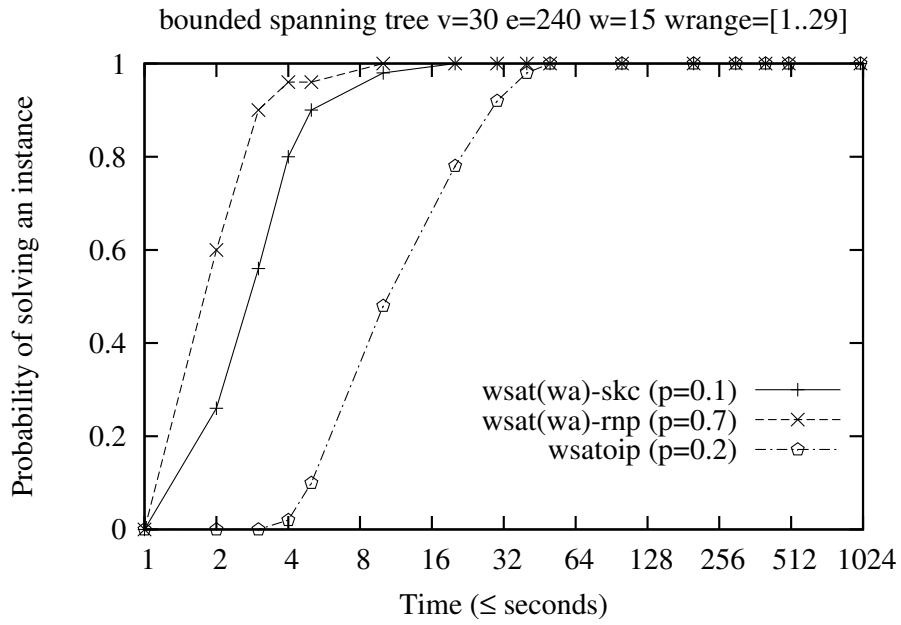

Figure C.4: $wsat(wa)$ v.s. $wsat(oip)$: bst

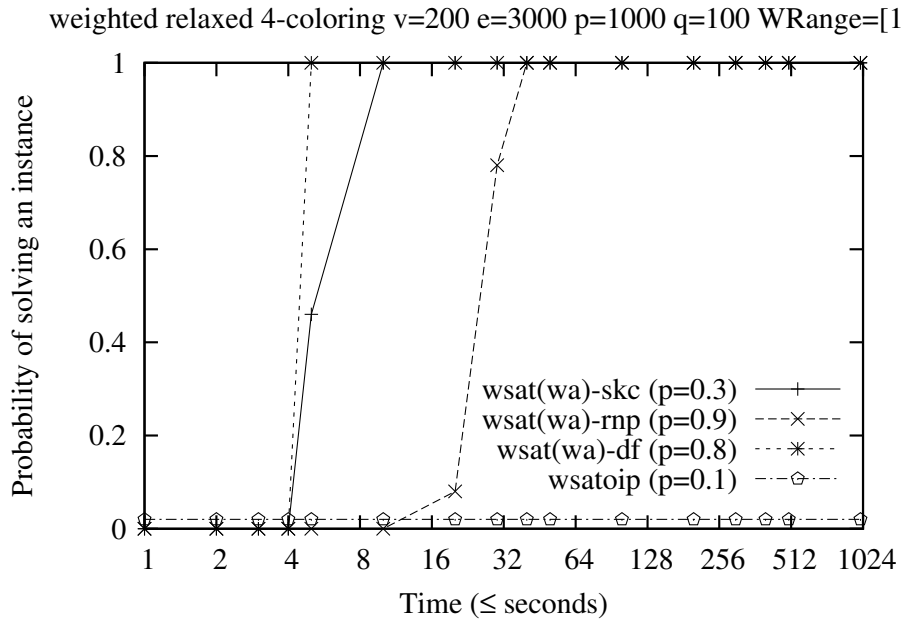

Figure C.5: $wsat(wa)$ v.s. $wsat(oip)$: $wrcol$

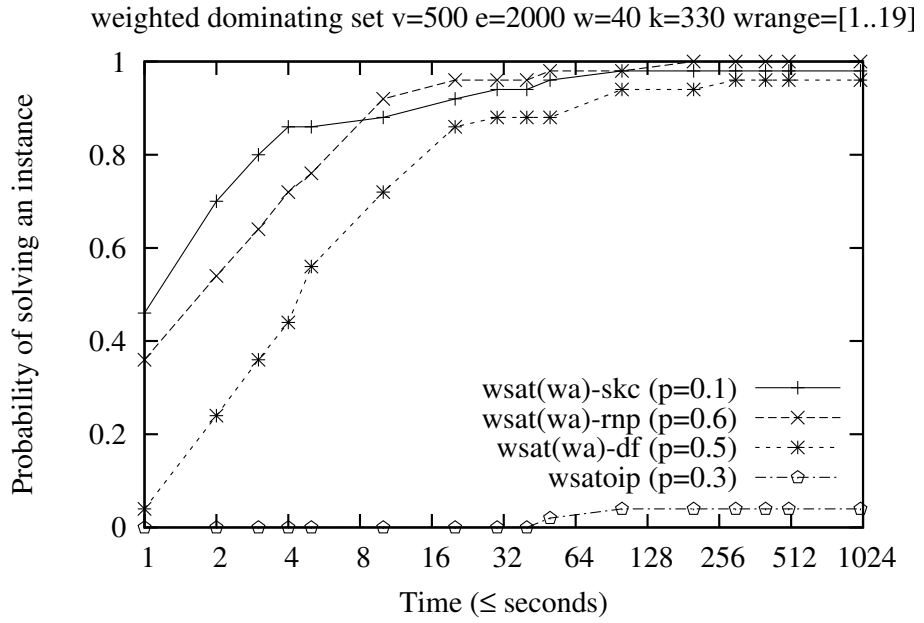

Figure C.6: $wsat(wa)$ v.s. $wsat(oip)$: wdm

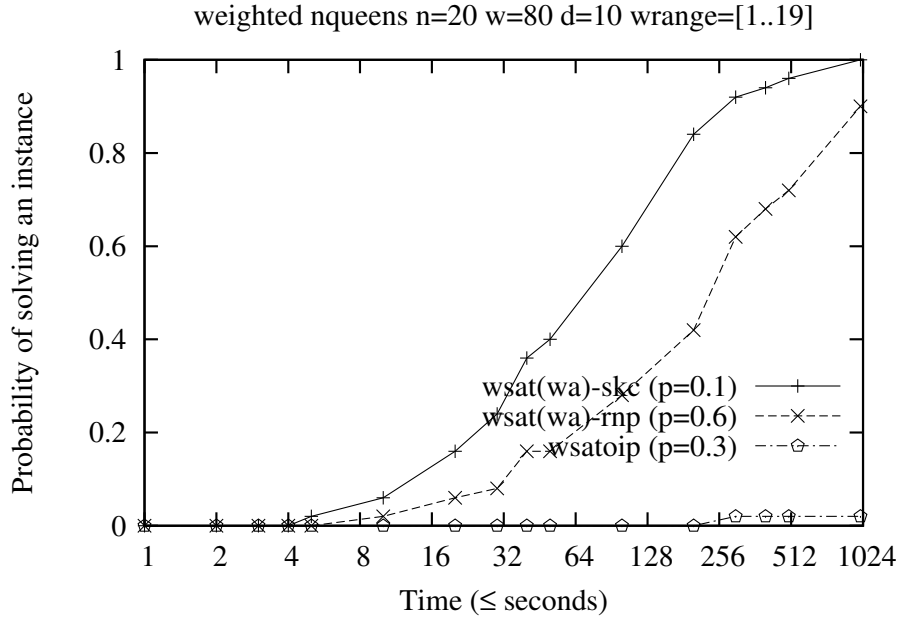

Figure C.7: $wsat(wa)$ v.s. $wsat(oip)$: $dwnq$

Appendix D Robustness w.r.t. the noise ratio

D.1 On *vcov* instances

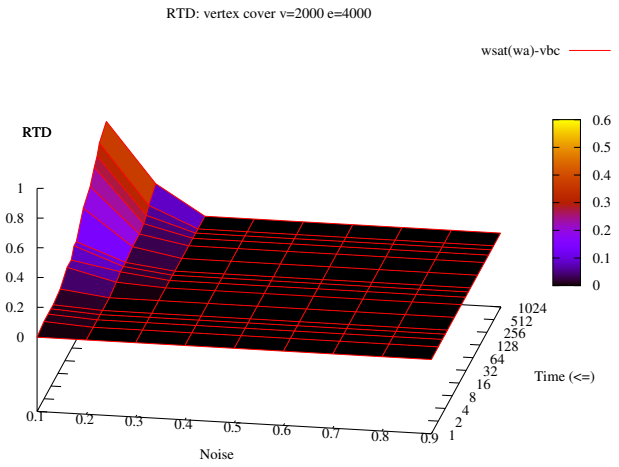

Figure D.1: *vcov*: $wsat(wa)$-skc

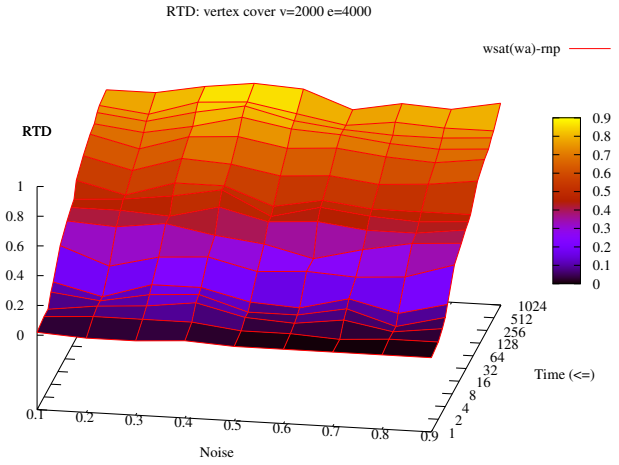

Figure D.2: *vcov*: $wsat(wa)$-rnp

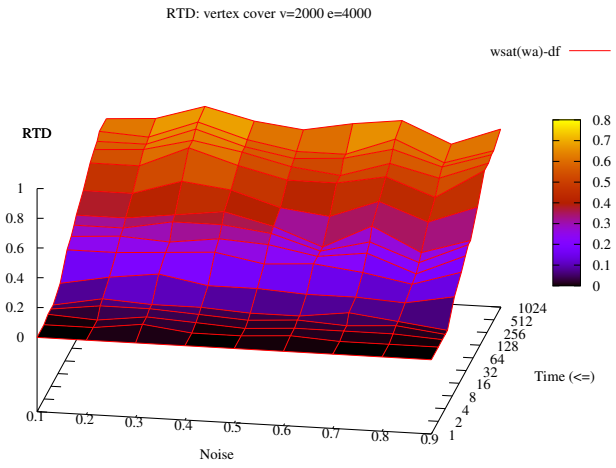

Figure D.3: $vcov$: $wsat(wa)$-df

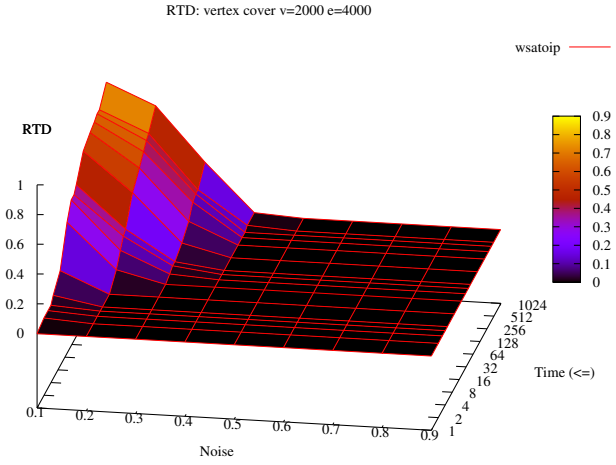

Figure D.4: $vcov$: $wsat(oip)$

D.2 On *wvcov* instances

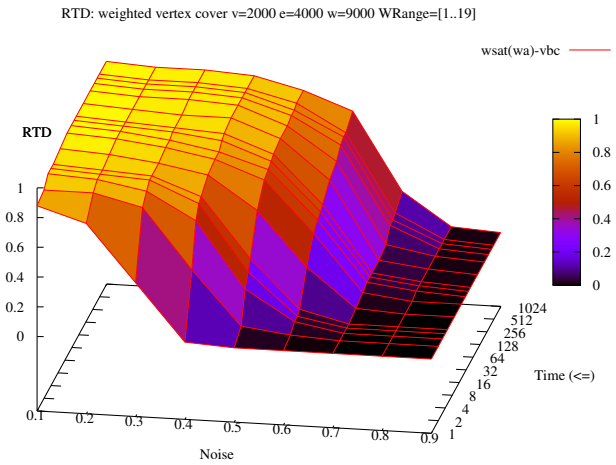

Figure D.5: *wvcov*: $wsat(wa)$-*skc*

D.3 On *tsp* instances

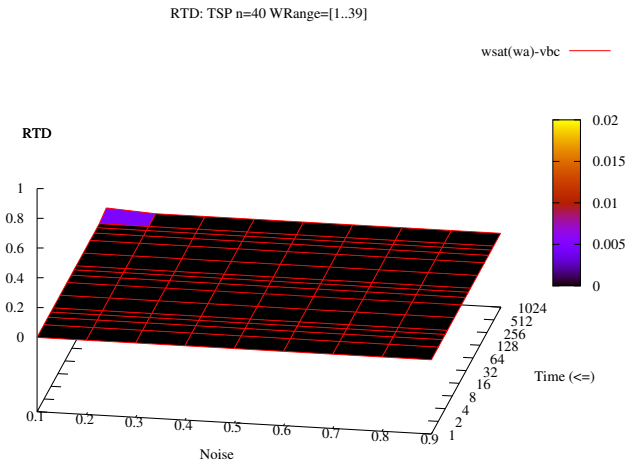

Figure D.6: *tsp*: $wsat(wa)$-*skc*

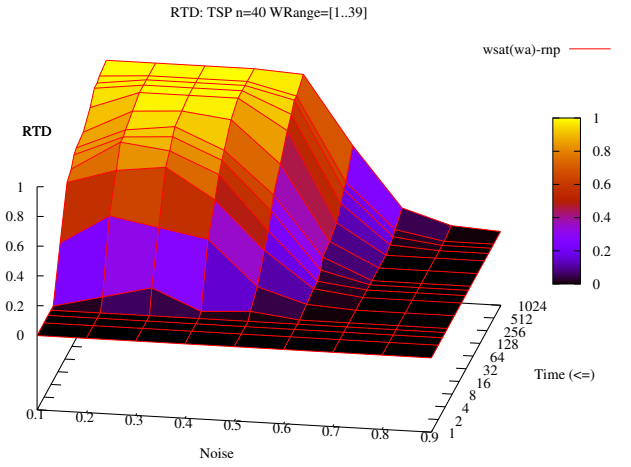

Figure D.7: *tsp*: $wsat(wa)$-*rnp*

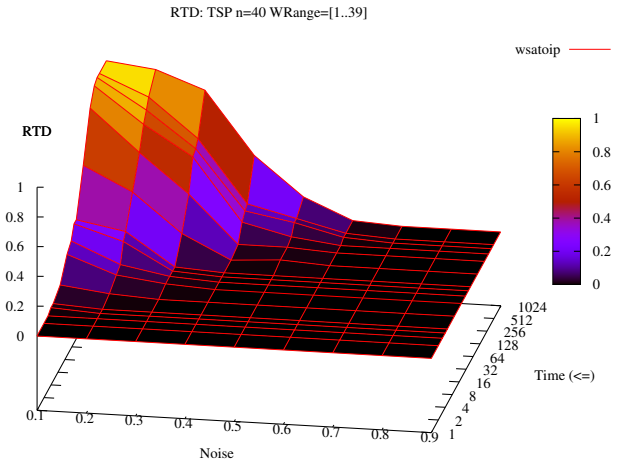

Figure D.8: tsp: $wsat(oip)$

D.4 On *bst* instances

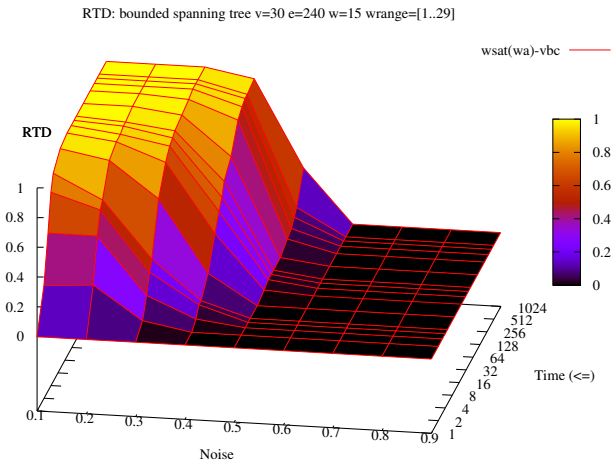

Figure D.9: *bst*: $wsat(wa)$-*skc*

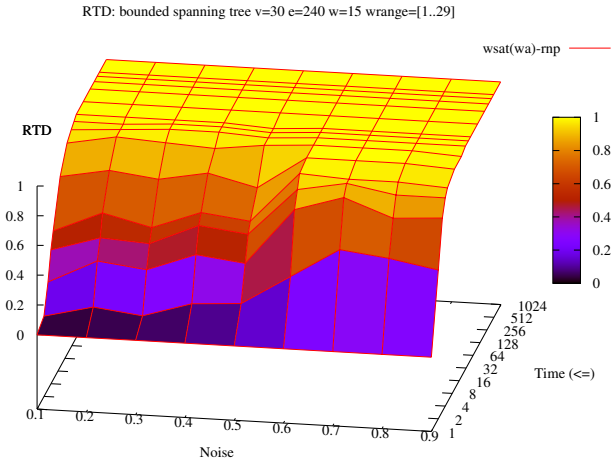

Figure D.10: *bst*: $wsat(wa)$-*rnp*

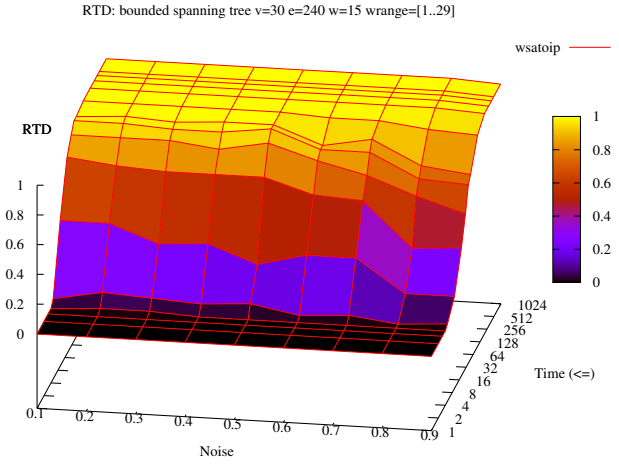

Figure D.11: bst: $wsat(oip)$

D.5 On *wrcol* instances

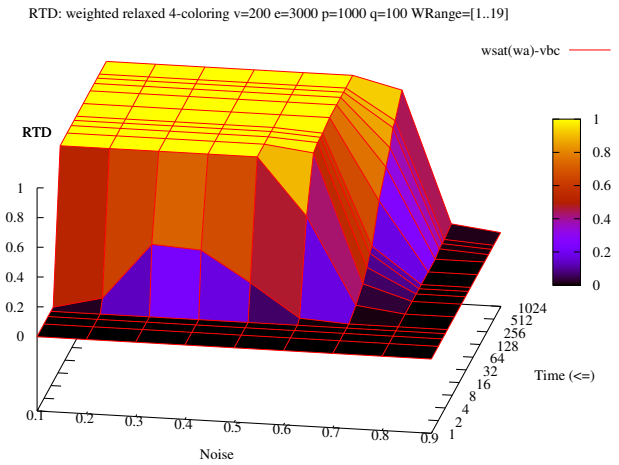

Figure D.12: *wrcol*: $wsat(wa)$-*skc*

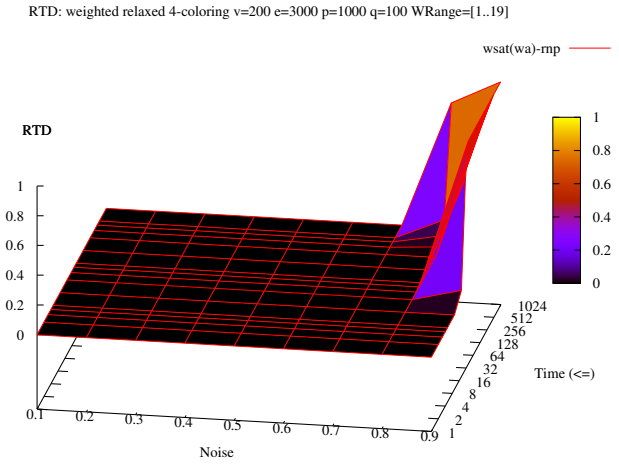

Figure D.13: *wrcol*: $wsat(wa)$-*rnp*

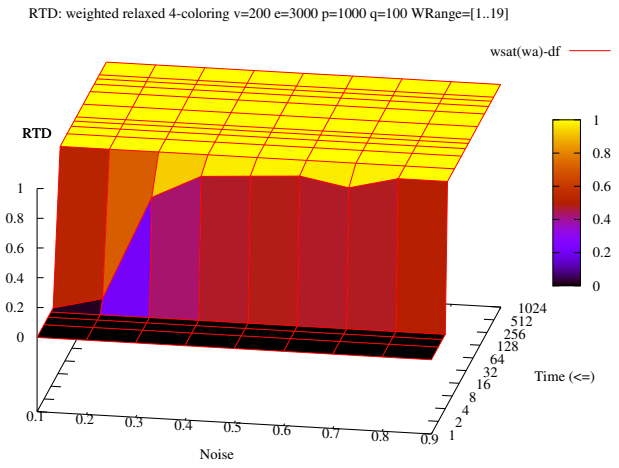

Figure D.14: $wrcol$: $wsat(wa)$-df

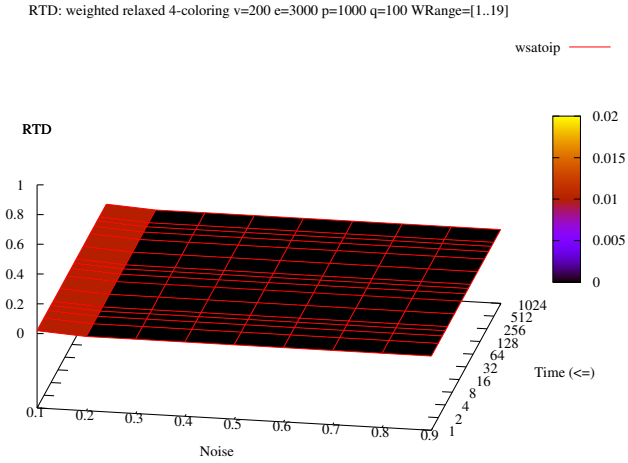

Figure D.15: $wrcol$: $wsat(oip)$

D.6 On wdm instances

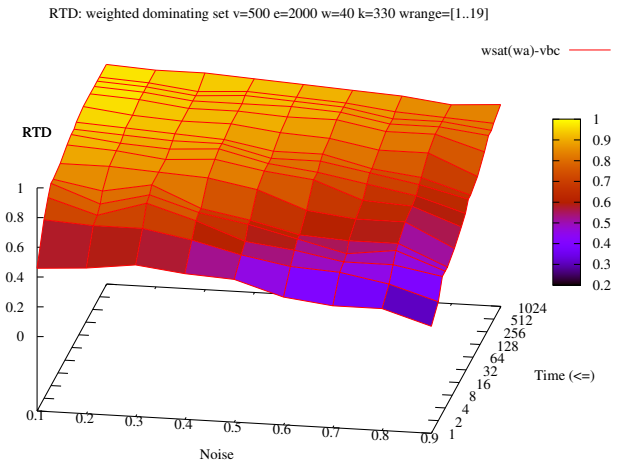

Figure D.16: wdm: $wsat(wa)$-skc

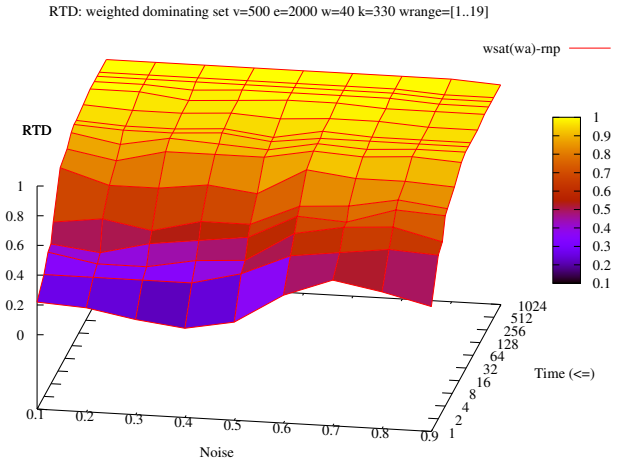

Figure D.17: wdm: $wsat(wa)$-rnp

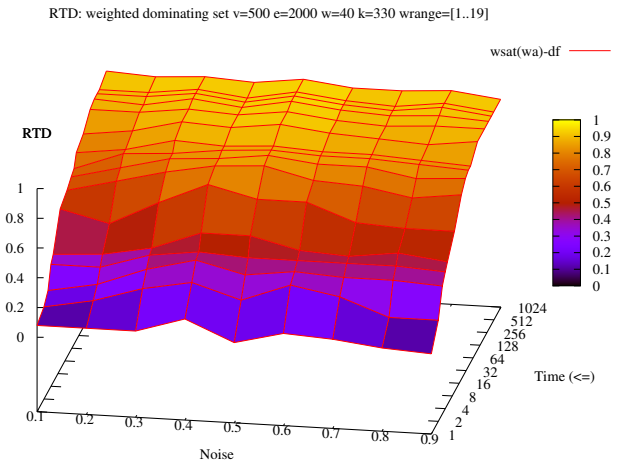

Figure D.18: wdm: $wsat(wa)$-df

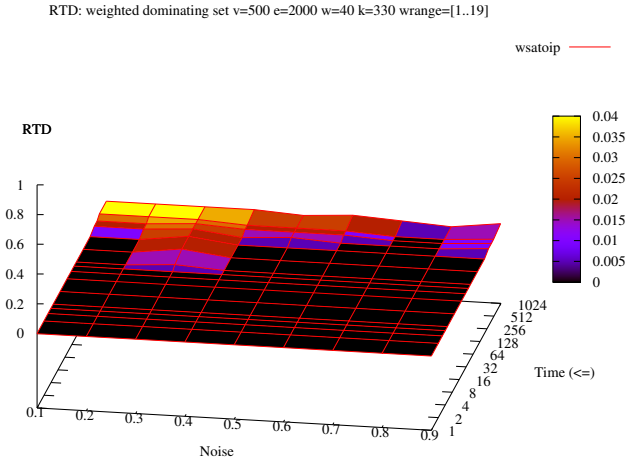

Figure D.19: wdm: $wsat(oip)$

D.7 On *dwnq* instances

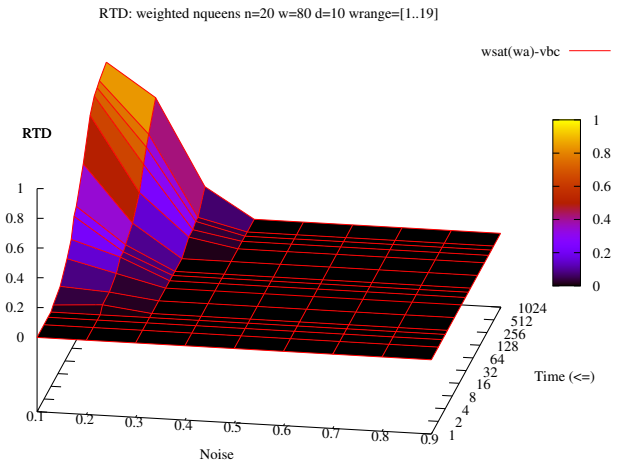

Figure D.20: *dwnq*: $wsat(wa)$-*skc*

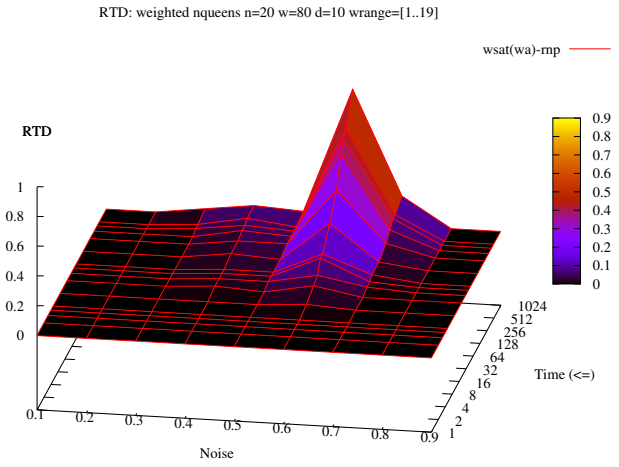

Figure D.21: *dwnq*: $wsat(wa)$-*rnp*

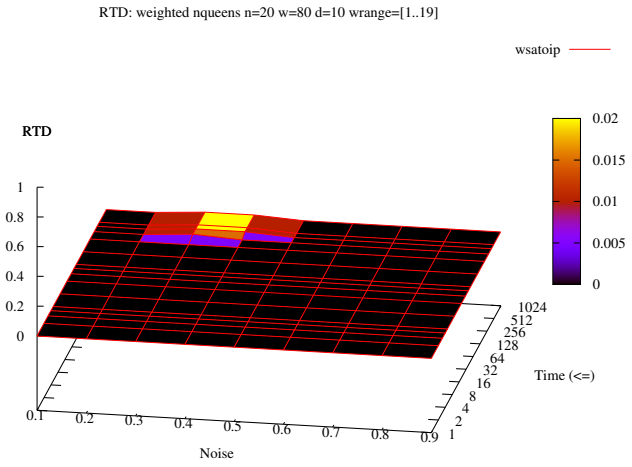

Figure D.22: $dwnq$: $wsat(oip)$

Bibliography

[1] F. ALOUL, A. RAMANI, I. MARKOV, AND K. SAKALLAH, *PBS: a backtrack-search pseudo-boolean solver and optimizer*, in Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability, 2002, pp. 346 – 353.

[2] F. ALOUL, A. RAMANI, I. MARKOV, AND K. SAKALLAH, *PBS v0.2, incremental pseudo-boolean backtrack search SAT solver and optimizer*, 2003. http://www.eecs.umich.edu/~faloul/Tools/pbs/.

[3] H. ANDRÉKA AND I. NÉMETI, *The generalized completeness of Horn predicate logic as a programming language*, Acta Cybernetica, 4 (1978/79), pp. 3–10.

[4] C. ANGER, K. KONCZAK, AND T. LINKE, *Nomore: Nonmonotonic reasoning with logic programs*, in Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA), vol. 2424, Lecture Notes in Computer Science, Springer, 2002, pp. 521–524.

[5] K. APT, *Logic programming*, in Handbook of theoretical computer science, J. van Leeuven, ed., Elsevier, Amsterdam, 1990, pp. 493–574.

[6] C. ARAVINDAN, J. DIX, AND I. NIEMELÄ, *DisLoP: Towards a disjunctive logic programming system*, in Logic Programming and Nonmonotonic Reasoning (Dagstuhl, Germany, 1997), vol. 1265 of Lecture Notes in Computer Science, Springer, 1997, pp. 342–353.

[7] Y. BABOVICH AND V. LIFSCHITZ, Cmodels package, 2002. http://www.cs.utexas.edu/users/tag/cmodels.html.

[8] C. BARAL AND M. GELFOND, *Logic programming and knowledge representation*, Journal of Logic Programming, 19(20) (1994), pp. 73–148.

[9] C. BARAL, M. GELFOND, AND A. PROVETTI, *Representing actions: laws, observations and hypotheses*, Journal of Logic Programming, 31 (1997), pp. 201–243.

[10] C. BARAL AND V. SUBRAHMANIAN, *Dualities between alternative semantics for logic programming and nonmonotonic reasoning (extended abstract)*, in Logic programming and non-monotonic reasoning (Washington, DC, 1991), A. Nerode, W. Marek, and V. Subrahmanian, eds., Cambridge, MA, 1991, MIT Press, pp. 69–86.

[11] P. BARTH, *A Davis-Putnam based elimination algorithm for linear pseudo-boolean optimization*, tech. rep., Max-Planck-Institut für Informatik, 1995. MPI-I-95-2-003.

[12] B. BENHAMOU, L. SAIS, , AND P. SIEGEL, *Two proof procedures for a cardinality based language in propositional calculus*, in Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS-1994), vol. 775 of LNCS, Springer, 1994, pp. 71–82.

[13] C. BOUTILIER, R. BRAFMAN, C. DOMSHLAK, H. HOOS, AND D. POOLE, *Cp-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements*, Journal of Artificial Intelligence Research, 21 (2003), pp. 135–191.

[14] ——, *Preference-based constrained optimization with cp-nets*, Computational Intelligence, 20 (2004), pp. 137–157.

[15] C. BOUTILIER, R. BRAFMAN, H. HOOS, AND D. POOLE, *Reasoning with conditional ceteris paribus preference statements*, in Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-99), 1999, pp. 71–80.

[16] R. BRAFMAN AND C. DOMSHLAK, *Introducing variable importance tradeoffs into cp-nets*, in Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence (UAI-02), Morgan Kaufmann, 2002.

[17] G. BREWKA, I. NIEMELÄ, AND M. TRUSZCZYŃSKI, *Answer set optimization*, in Proceedings of the 18th International Joint Conference on Artificial Intelligence, 2003, pp. 867–872.

[18] K. CLARK, *Negation as failure*, in Logic and data bases, H. Gallaire and J. Minker, eds., Plenum Press, New York-London, 1978, pp. 293–322.

[19] M. DAVIS, G. LOGEMANN, AND D. LOVELAND, *A machine program for theorem-proving*, Comm. Assoc. for Computing Machinery, 5 (1962), pp. 394–397.

[20] T. DELL'ARMI, W. FABER, G. IELPA, N. LEONE, S. PERRI, AND G. PFEIFER, *System description: DLV with aggregates*, in Logic programming and Nonmonotonic Reasoning, Proceedings of the t^{th} International Conference, V. Lifschitz and I. Niemelä, eds., vol. 2923, Springer, 2004, pp. 326–330.

[21] T. DELL'ARMI, W. FABER, G. IELPA, N. LEONE, AND G. PFEIFER, *Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV*, in Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-2003), Morgan Kaufmann, 2003, pp. 847–852.

[22] M. DENECKER, V. MAREK, AND M. TRUSZCZYŃSKI, *Approximations, stable operators, well-founded fixpoints and applications in nonmonotonic reasoning*, in Logic-Based Artificial Intelligence, J. Minker, ed., Kluwer Academic Publishers, 2000, pp. 127–144.

[23] ——, *Unified semantic treatment of default and autoepistemic logics*, in Principles of Knowledge Representation and Reasoning, Proceedings of the 7th International Conference (KR2000), Morgan Kaufmann Publishers, 2000, pp. 74 – 84.

[24] ——, *Ultimate approximations in nonmonotonic knowledge representation systems*, in Principles of Knowledge Representation and Reasoning, Proceedings of the 8th International Conference (KR2002), Morgan Kaufmann Publishers, 2002, pp. 177–188.

[25] ——, *Uniform semantic treatment of default and autoepistemic logics*, Artificial Intelligence Journal, 143 (2003), pp. 79–122.

[26] M. DENECKER, N. PELOV, AND M. BRUYNOOGHE, *Ultimate well-founded and stable semantics for logic programs with aggregates*, in Logic programming, Proceedings of the 2001 International Conference on Logic Programming, P. Codognet, ed., vol. 2237, Springer, 2001, pp. 212–226.

[27] Y. DIMOPOULOS AND A. SIDERIS, *Towards local search for answer sets*, in Proceedings of the 18th International Conference on Logic Programming, vol. 2401 of LNCS, Springer, 2002, pp. 363 – 367.

[28] H. DIXON AND M. GINSBERG, *Inference methods for a pseudo-boolean satisfiability solver*, in The 18th National Conference on Artificial Intelligence (AAAI-2002), AAAI Press, 2002, pp. 635–640.

[29] K. DOETS, *From Logic to Logic Programming*, Foundations of Computing Series, MIT Press, Cambridge, MA, 1994.

[30] W. DOWLING AND J. GALLIER, *Linear-time algorithms for testing the satisfiability of propositional Horn formulae*, Journal of Logic Programming, 1 (1984), pp. 267–284.

[31] M. DRANSFIELD, L. LIU, V. MAREK, AND M. TRUSZCZYŃSKI, *Satisfiability and computing van der waerden numbers*, The Electronic Journal of Combinatorics, 11(1) (2004).

[32] M. DRANSFIELD, V. MAREK, AND M. TRUSZCZYŃSKI, *Satisfiability and computing van der waerden numbers*, in Theory and Applications of Satisfiability Testing, 6th International Conference, SAT-2003, vol. 2919 of LNCS, Springer, 2003, pp. 1–13.

[33] D. EAST, M. IAKHIAEV, A. MIKITIUK, AND M. TRUSZCZYŃSKI, *Tools for modeling and solving search problems*, 2004. Submitted for publication (available at http://www.cs.uky.edu/psgrnd/).

[34] ——, *Tools for modeling and solving search problems*, 2006. submitted.

[35] D. EAST, L. LIU, S. LOGSDON, V. MAREK, AND M. TRUSZCZYŃSKI, *ASPPS user's manual*, 2003. http://www.cs.uky.edu/aspps/users_manual.ps.

[36] D. EAST AND M. TRUSZCZYŃSKI, *On the accuracy and running time of gsat*, in Proceedings of the 9th Portuguese Conference on Artificial Intelligence(EPIA'99), vol. 1695 of Lecture Notes in Artificial Intelligence, Springer, 1999, pp. 49–61.

[37] D. EAST AND M. TRUSZCZYŃSKI, *Datalog with constraints*, in Proccedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), AAAI Press, 2000, pp. 163–168.

[38] D. EAST AND M. TRUSZCZYŃSKI, *ASP solver* aspps, 2001. http://www.cs.uky.edu/aspps/.

[39] D. EAST AND M. TRUSZCZYŃSKI, aspps — *an implementation of answer-set programming with propositional schemata*, in Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001, vol. 2173, Lecture Notes in Artificial Intelligence, Springer, 2001, pp. 402–405.

[40] ——, *More on wire-routing*, in Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Press, 2001. Papers from the 2001 AAAI Spring Symposium, Technical Report SS-01-01.

[41] D. EAST AND M. TRUSZCZYŃSKI, *Propositional satisfiability in answer-set programming*, in Proceedings of Joint German/Austrian Conference on Artificial Intelligence (KI-2001), vol. 2174 of LNAI, Springer, 2001, pp. 138–153.

[42] D. EAST AND M. TRUSZCZYŃSKI, *The* aspps *system*, in Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA), vol. 2424, Lecture Notes in Computer Science, Springer, 2002, pp. 533–536.

[43] D. EAST AND M. TRUSZCZYŃSKI, *Propositional satisfiability in declarative programming*, 2003. http://xxx.lanl.gov/abs/cs.LO/0211033.

[44] D. EAST AND M. TRUSZCZYŃSKI, *Predicate-calculus based logics for modeling and solving search problems*, ACM Transactions on Computational Logic, (2004). To appear, available at http://www.acm.org/tocl/accepted.html.

[45] N. EÉN AND N. SÖRENSSON, *An extensible SAT solver*, in Theory and Applications of Satisfiability Testing, 6th International Conference, SAT-2003, vol. 2919 of LNCS, Springer, 2003, pp. 502–518.

[46] T. EITER, W. FABER, N. LEONE, AND G. PFEIFER, *Declarative problem-solving in DLV*, in Logic-Based Artificial Intelligence, J. Minker, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 79–103.

[47] T. EITER AND M. FINK, *Uniform equivalence of logic programs under the stable model semantics*, in Proceedings of the 2003 International Conference on Logic Programming, vol. 2916 of Lecture Notes in Computer Science, Berlin, 2003, Springer, pp. 224–238.

[48] E. ERDEM AND V. LIFSCHITZ, *Tight logic programs*, Theory and Practice of Logic Programming, 3 (2003), pp. 499–518.

[49] W. FABER, N. LEONE, AND G. PFEIFER., *Recursive aggregates in disjunctive logic programs: Semantics and complexity.*, in Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004), number 3229 in Lecture Notes in AI (LNAI)., 2004, pp. 200 – 212.

[50] F. FAGES, *Consistency of Clark's completion and existence of stable models*, Journal of Methods of Logic in Computer Science, 1 (1994), pp. 51–60.

[51] P. FERRARIS, *Answer sets for propositional theories*. http://www.cs.utexas.edu/~otto/papers/proptheories.ps, 2004.

[52] P. FERRARIS AND V. LIFSCHITZ, *Weight constraints ans nested expressions*, Theory and Practice of Logic Programming, (forthcoming), (2004).

[53] R. FINKEL, V. MAREK, AND M. TRUSZCZYŃSKI, *Constraint lingo: A program for solving logic puzzles and other tabular constraint problems*, in Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA), vol. 2424, Lecture Notes in Computer Science, Springer, 2002, pp. 513–516.

[54] ——, *Constraint lingo: A program for modeling and solving tabular constraint problems*. Submitted for journal publication, 2003.

[55] R. FINKEL, V. MAREK, AND M. TRUSZCZYŃSKI, *Constraint lingo: Towards high-level constraint programming*, Software Practice and Experience, 34 (2004), pp. 1481–1504.

[56] R. FINKEL, W. MAREK, AND M. TRUSZCZYNSKI, *Tabular constraint-satisfaction problems and answer-set programming*, in Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Press, 2001. Papers from the 2001 AAAI Spring Symposium, Technical Report SS-01-01.

[57] M. C. FITTING, *Fixpoint semantics for logic programming – a survey*, Theoretical Computer Science, 278 (2002), pp. 25–51.

[58] M. GAREY AND D. JOHNSON, *Computers and intractability. A guide to the theory of NP-completeness*, W.H. Freeman and Co., San Francisco, Calif., 1979.

[59] M. GELFOND AND V. LIFSCHITZ, *The stable semantics for logic programs*, in Proceedings of the 5th International Conference on Logic Programming, MIT Press, 1988, pp. 1070–1080.

[60] M. GINSBERG, ed., *Readings in Nonmonotonic Reasoning*, Morgan Kaufmann, Palo Alto, CA, 1987.

[61] M. GINSBERG AND D. MCALLESTER, *GSAT and dynamic bactracking*, in Principles of Knowledge Representation and Reasoning, KR '94, J. Doyle, E. Sandewall, and P. Torasso, eds., Morgan Kaufmann, 1994, pp. 226–237.

[62] E. GIUNCHIGLIA, Y. LIERLER, AND M. MARATEA, *SAT-based answer-set programming*, in Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-2004), AAAI Press, 2004, pp. 61–66.

[63] E. GOLDBERG AND Y. NOVIKOV, *Berkmin: a fast and robust sat-solver*, in DATE-2002, 2002, pp. 142–149.

[64] C. GOMES, B. SELMAN, N. CRATO, AND H. KAUTZ, *Heavy-tailed phenomena in satisfiability and constraint satisfaction problems*, in SAT2000: Highlights of satisfiability research in the year 2000, I. Gent, H. van Maaren, and T. Walsh, eds., IOS Press, 2000, pp. 15–41.

[65] J. HANSEN AND B. JAUMARD, *Algorithms for the maximum satisfiability problem*, Computing, 44 (1990), pp. 279–303.

[66] K. HELJANKO AND I. NIEMELÄ, *Bounded ltl model checking with stable models*, Theory and Practice of Logic Programming (TPLP), 3(4,5) (2003), pp. 519–550.

[67] J. HOOKER, *Logic-Based Methods for Optimization*, J. Wiley and Sons, 2000.

[68] H. HOOS, *On the run-time behaviour of stochastic local search algorithms for sat*, in Proceedings of The Sixteenth National Conference on Artificial Intelligence (AAAI-99), Orlando, Florida, 1999, pp. 661–666.

[69] ——, *An adaptive noise mechanism for walksat*, in Proceedings of The Nineteenth National Conference on Artificial Intelligence (AAAI-99), 2002, pp. 655–660.

[70] H. HOOS AND T. STÜTZLE, *Local search algorithms for sat: An empirical evaluation*, in SAT2000: Highlights of Satisfiability REsearch in the Year 2000, I. Gent, H. van Maaren, and T. Walsh, eds., vol. 62 of Frontiers in Artificial Intelligence and Applications, IOS Press, Amsterdam, 2000, pp. 43–88.

[71] H. H. HOOS AND T. STÜTZLE, *A characterization the run-time behaviour of stochastic local search*. http://www.cs.ubc.ca/~hoos/Publ/aida-98-01.ps, 1998.

[72] ——, *Stochastic Local Search Foundations and Applications*, Morgan Kaufmann, San Francisco, CA, USA, 2004.

[73] H. KAUTZ, *Blackbox — a sat technology planning system*, 2003. http://www.cs.washington.edu/homes/kautz/satplan/blackbox/.

[74] H. KAUTZ, D. MCALLESTER, AND B. SELMAN, *Encoding plans in propositional logic*, in Proceedings of 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-1996), Morgan Kaufmann, 1996, pp. 374–384.

[75] N. LEONE, G. PFEIFER, W. FABER, T. EITER, G. GOTTLOB, S. PERRI, AND F. SCARCELLO, *The dlv system for knowledge representation and reasoning*, ACM Transactions on Computational Logic, (2004). To appear, available at http://xxx.lanl.gov/abs/cs.AI/0211004.

[76] N. LEONE, G. PFEIFER, AND F. W, *System* dlv, 1997. http://www.dbai.tuwien.ac.at/proj/dlv/.

[77] C. LI, *SAT solver* satz, 1997. http://www.laria.u-picardie.fr/~cli/EnglishPage.html.

[78] ——, *Integrating equivalency reasoning into davis-putnam procedure*, in Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000), AAAI Press, 2000, pp. 291–296.

[79] V. LIFSCHITZ, D. PEARCE, AND A. VALVERDE, *Strongly equivalent logic programs*, ACM Transactions on Computational Logic, 2(4) (2001), pp. 526–541.

[80] F. LIN, *Reducing strong equivalence of logic programs to entailment in classical propositional logic*, in Principles of Knowledge Representation and Reasoning, Proceedings of the 8th International Conference (KR2002), Morgan Kaufmann Publishers, 2002.

[81] F. LIN AND Y. ZHAO, *ASSAT: Computing answer sets of a logic program by SAT solvers*, in Proccedings of the 18th National Conference on Artificial Intelligence (AAAI-2002), AAAI Press, 2002, pp. 112–117.

[82] L. LIU AND M. TRUSZCZYŃSKI, *Local-search techniques in propositional logic extended with cardinality atoms*, in Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003), F. Rossi, ed., vol. 2833 of LNCS, Springer, 2003, pp. 495–509. Lecture Notes in Computer Science, Springer.

[83] ——, *Local search with bootstrapping*, in Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT-2004), 2004.

[84] ——, Wsat(cc) — *a fast local-search ASP solver*, in Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7), vol. 2923 of LNCS, Springer, 2004, pp. 351–355. Lecture Notes in Computer Science, Springer.

[85] ——, *Pbmodels — software to compute stable models by pseudoboolean solvers*, in Proceedings of the 8th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR-05), vol. 3662 of LNCS, Springer, 2005, pp. 410–415. Lecture Notes in Computer Science, Springer.

[86] ——, *Properties of programs with monotone and convex constraints*, in Proceedings of The Twentieth National Conference on Artificial Intelligence (AAAI-05), AAAI Press, 2005, pp. 701–706.

[87] ——, *Local search techniques for boolean combinations of pseudo-boolean constarints*, in Proceedings of The Twenty First National Conference on Artificial Intelligence (AAAI-06), AAAI Press, 2006, p. Accepted.

[88] ——, *Properties and applications of programs with monotone and convex constraints*, The Journal of Artificial Intelligence Research, Accepted (2006).

[89] Z. LONC AND M. TRUSZCZYŃSKI, *On the problem of computing the well-founded semantics*, in Proceedings of the 1st International Conference on Computational Logic, CL-2000, Springer, 2000, pp. 673–687. Lecture Notes in Artificial Intelligence, Vol. 1861.

[90] ——, *Fixed-parameter complexity of semantics for logic programs*, in Logic programming, Proceedings of the 2001 International Conference on Logic Programming, vol. 2237 of Lecture Notes in Computer Science, Springer, 2001, pp. 197–211.

[91] ——, *On the problem of computing the well-founded semantics*, Theory and Practice of Logic Programming, 5 (2001), pp. 591–609.

[92] ——, *Computing stable models: worst-case performance estimates*, in Logic Programming, Proceedings of the 2002 International Conference on Logic Programming, vol. 2401 of Lecture Notes in Computer Science, Springer, 2002, pp. 347–362.

[93] ——, *Computing minimal models, stable models and answer sets*, in Logic Programming, Proceedings of the 2003 International Conference on Logic Programming, Lecture Notes in Computer Science, Springer, 2003.

[94] ——, *Computing stable models: worst-case performance estimates*, Theory and Practice of Logic Programming, (2003). To appear.

[95] ——, *Fixed-parameter complexity of semantics for logic programs*, ACM Transactions on Computational Logic, (2003). To appear.

[96] V. MANQUINHO AND O. ROUSSEL, *Pseudo boolean evaluation 2005*, 2005. http://www.cril.univ-artois.fr/PB05/.

[97] V. MAREK, I. NIEMELÄ, AND M. TRUSZCZYŃSKI, *Characterizing stable models of logic programs with cardinality constraints*, in Proceedings of the 7th Interna-

tional Conference on Logic Programming and Nonmonotonic Reasoning, vol. 2923 of Lecture Notes in Artificial Intelligence, Springer, 2004, pp. 154–166.

[98] V. MAREK AND J. REMMEL, *Logic programs with cardinality constraints*, in Proceedings of the 9th International Workshop on Nonmonotonic Reasoning, 2002, pp. 219–228.

[99] ——, *Set constraints in logic programming*, in Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, 2004. Lecture Notes in Computer Science, Springer.

[100] ——, *On body-normal programs with cardinality constraints*. Unpublished manuscript, 2005.

[101] V. MAREK AND M. TRUSZCZYŃSKI, *Stable models and an alternative logic programming paradigm*, in The Logic Programming Paradigm: a 25-Year Perspective, K. Apt, W. Marek, M. Truszczyński, and D. Warren, eds., Springer, Berlin, 1999, pp. 375–398.

[102] ——, *Logic programs with abstract constraint atoms*, in Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), AAAI Press, 2004, pp. 86–91.

[103] W. MAREK, A. NERODE, AND M. TRUSZCZYNSKI, eds., *Logic programming and nonmonotonic reasoning. Proceedings of the 3rd International Conference (LPNMR '95) held in Lexington, KY, June 26–28, 1995*, vol. 928 of Lecture Notes in Computer Science, Springer, Berlin, 1995.

[104] W. MAREK AND J. REMMEL, *On the foundations of answer-set programming*, in Answer-Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning, AAAI Press, 2001. Papers from the 2001 AAAI Spring Symposium, Technical Report SS-01-01.

[105] J. MARQUES-SILVA AND K. SAKALLAH, *GRASP: A new search algorithm for satisfiability*, IEEE Transactions on Computers, 48 (1999), pp. 506–521.

[106] M. MOSKEWICZ, C. MADIGAN, Y. ZHAO, L. ZHANG, AND S. MALIK, *Chaff: engineering an efficient SAT solver*, in Proceedings of the 38th ACM IEEE Design Automation Conference, ACM Press, 2001, pp. 530–535.

[107] ——, *SAT solver* chaff, 2001. http://www.ee.princeton.edu/~chaff/.

[108] I. NIEMELÄ, *Logic programs with stable model semantics as a constraint programming paradigm*, in Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning, I. Niemelä and T. Schaub, ed., 1998, pp. 72–79.

[109] ——, *Logic programming with stable model semantics as a constraint programming paradigm*, Annals of Mathematics and Artificial Intelligence, 25 (1999), pp. 241–273.

[110] I. NIEMELÄ AND P. SIMONS, *Smodels — an implementation of the stable model and well-founded semantics for normal logic programs*, in Logic Programming and Nonmonotonic Reasoning (the 4th International Conference, Dagstuhl, Germany, 1997), vol. 1265 of Lecture Notes in Computer Science, Springer, 1997, pp. 420–429.

[111] ——, *Extending the smodels system with cardinality and weight constraints*, in Logic-Based Artificial Intelligence, J. Minker, ed., Kluwer Academic Publishers, 2000, pp. 491–521.

[112] I. NIEMELÄ AND P. SIMONS, *Extending the Smodels system with cardinality and weight constraints*, in Logic-Based Artificial Intelligence, J. Minker, ed., Kluwer Academic Publishers, Dordrecht, 2000, pp. 491–521.

[113] I. NIEMELÄ, P. SIMONS, AND T. SOININEN, *Stable model semantics of weight constraint rules*, in Proceedings of LPNMR-1999, vol. 1730 of Lecture Notes in Computer Science, Springer, 1999, pp. 317–331.

[114] I. NIEMELÄ, P. SIMONS, AND T. SYRJÄNEN, *SLP solver* smodels, 1997. http://www.tcs.hut.fi/Software/smodels/.

[115] I. NIEMELÄ, P. SIMONS, AND T. SYRJÄNEN, *Smodels: a system for answer set programming*, in Proceedings of the 8th International Workshop on Non-Monotonic Reasoning, NMR'2000, Breckenridge, Co., C. Baral and M. Truszczyński, eds., 2000.

[116] A. PARKES, *Lifted Search Engines for Satisfiability*, PhD thesis, University of Oregon, Department of Computer Science, 1999.

[117] N. PELOV., *Semantics of logic programs with aggregates*, PhD Thesis. Department of Computer Science, K.U.Leuven, Leuven, Belgium, (2004).

[118] N. PELOV, M. DENECKER, AND M. BRUYNOOGHE, *Partial stable models for logic programs with aggregates*, in Logic programming and Nonmonotonic Reasoning, Proceedings of the 7^{th} International Conference, V. Lifschitz and I. Niemelä, eds., vol. 2923, Springer, 2004, pp. 207–219.

[119] N. PELOV, M. DENECKER, AND M. BRUYNOOGHE, *Partial stable semantics for logic programs with aggregates*, in Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, V. Lifschitz and I. Niemelä, eds., vol. 2923 of Lecture Notes in Artificial Intelligence, Springer, 2004, pp. 207–219.

[120] S. PRESTWICH, *Randomised backtracking for linear pseudo-boolean constraint problems*, in Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, (CPAIOR-2002), 2002, pp. 7–20. http://www.emn.fr/x-info/cpaior/Proceedings/CPAIOR.pdf.

[121] S. PRESWITCH, *Randomised backtracking for weightless linear pseudo-boolean constraint problems*, in Fourth International Workshop on Integration of AI and OR

techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR'02), N. Jussien and F. Laburthe, eds., Le Croisic, France, 2002, pp. 7–19.

[122] B. SELMAN, *Stochastic search and phase transitions: AI meets physics*, in Proceedings of IJCAI-95, Morgan Kaufmann, 1995, pp. 998–1002.

[123] B. SELMAN, H. KAUTZ, AND B. COHEN, *Noise strategies for improving local search*, in Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-1994), Seattle, USA, 1994, AAAI Press, pp. 337–343.

[124] B. SELMAN AND H. A. KAUTZ, *Planning as satisfiability*, in Proceedings of the 10th European Conference on Artificial Intelligence, Vienna, Austria, 1992.

[125] P. SIMONS, I. NIEMELÄ, AND T. SOININEN, *Extending and implementing the stable model semantics*, Artificial Intelligence, 138 (2002), pp. 181–234.

[126] T. SYRJÄNEN, lparse*, a procedure for grounding domain restricted logic programs*. http://www.tcs.hut.fi/Software/smodels/lparse/, 1999.

[127] T. SYRJÄNEN AND I. NIEMELÄ, *Cardinality constraints, variables and stable models*, (2002). A manuscript.

[128] H. TURNER, *Strong equivalence for logic programs and default theories (made easy)*, in Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001, vol. 2173, Lecture Notes in Artificial Intelligence, Springer, 2001, pp. 81–92.

[129] ——, *Strong equivalence made easy: Nested expressions and weight constraints*, Theory and Practice of Logic Programming, 3, (4&5) (2003), pp. 609–622.

[130] M. VAN EMDEN AND R. KOWALSKI, *The semantics of predicate logic as a programming language*, Journal of the ACM, 23 (1976), pp. 733–742.

[131] J. WALSER, *SLS PB solver* wsatoip, 1997. http://www.ps.uni-sb.de/~walser/wsatpb/wsatpb.html.

[132] J. WALSER, *Solving linear pseudo-boolean constraints with local search*, in Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-97), AAAI Press, 1997, pp. 269–274.

[133] K. XU, *Bhoslib*, 2005. URL: http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm.

[134] H. ZHANG, *SATO: an efficient propositional prover*, in Proceedings of the International Conference on Automated Deduction (CADE-97), 1997, pp. 308–312. Lecture Notes in Artificial Intelligence, 1104.

[135] ——, *Generating college conference basketball schedules by a sat solver*, in Proceedings of the Fifth International Symposium on Theory and Applications of Satisfiability Testing (SAT-2002), 2002, pp. 281–291.

Vita

1. Background.

   (a) Date of Birth: 06 March, 1975

   (b) Place of Birth: Changchun, Jilin, China

2. Academic Degrees.

   (a) Ph.D., August 2006 (expected)

   Computer Science Department, University of Kentucky, Lexington, Kentucky, U.S.A.

   (b) M.E., Date received: July 2000

   Institute of Mathematics, Chinese Academy of Science, Beijing, China

   (c) B.S., Date received: July 1997

   Computer Science Department, Jilin University, Changchun, Jilin, China

3. Professional Experience.

   (a) Research Assistant, Spring 2001 - present

   Computer Science Department, University of Kentucky, Lexington, Kentucky

   Adviser: Dr. Miroslaw Truszczynski

   Research: knowledge representation and reasoning, logic programming, answer-set programming, SAT, CSP, and planning under uncertainty

   Major Projects:

   - Welfare to Work project (NSF grant ITR-0325063). Responsibilities include modeling application domain using factored MDPs, developing planning algorithms under uncertainty with hard and soft constraints

- Theoretical and practical aspects of answer-set programming formalisms (NSF grant IIS-0097278). Responsibilities include designing and implementing local search algorithms for propositional logic theories with weight atoms, extending properties of normal logic programs to monotone-constraint programs, implementing algorithms to compute stable models of logic programs with weight constraints (an instantiation of monotone-constraint programs)

(b) Instructor, Fall 2003

Computer Science Department, University of Kentucky, Lexington, Kentucky

CS375: *Logic and Theory of Computing*. Full responsibilities including lecture preparation and presentation, homework assignments and exams

(c) Instructor, Summer 2001

Computer Science Department, University of Kentucky, Lexington, Kentucky

CS216: *Introduction to Software Engineering*. Full responsibilities including lecture preparation and presentation, homework assignments and exams

(d) Instructor, Fall 2000 and Spring 2001

CS221: *FORTRAN Programming Language*. Major responsibilities including lecture preparation and presentation

(e) Research Assistant, Fall 1998 — Fall 2000

Institute of Mathematics, Academia Sinica, Beijing, China

Adviser: Dr. Ruqian Lu

Research: natural language processing, common-sense knowledge representation and reasoning

Project: *Pragmatic Aspects of Common-sense Knowledge*, supported by Chinese Natural Science Foundation

(f) Research Assistant, Spring 1998

Institute of Mathematics, Academia Sinica, Beijing, China

Adviser: Dr. Ruqian Lu

Research: 3D computer animation and virtual reality

Project: *Virtual Dentist Training System*, supported by Chinese Natural Science Foundation

(g) Research Assistant, Fall 1997

Institute of Mathematics, Academia Sinica, Beijing, China

Adviser: Dr. Ruqian Lu

Research: computer animation generation, and script analysis

Project: *Full-Life-Cycle Automation of Computer Animation*, supported by Chinese Natural Science Foundation

4. Publications.

(a) Papers in Refereed Journals

i. *Satisfiability and computing van der Waerden numbers*
Michael R. Dransfield, Lengning Liu, Victor W. Marek and Miroslaw Truszczynski
In *the Electronic Journal of Combinatorics*, Volume 11, 2004

(b) Papers at Refereed Conferences

i. *Pbmodels - software to compute stable models by pseudoboolean solvers*
Lengning Liu and Miroslaw Truszczynski
In *Proceedings of the 8th International Conference on Logic Programming and Non Monotonic Reasoning*, September, 2005, Diamante, Italy. Page 410 - 415, LNCS 3662, Springer

ii. *Properties of logic programs with monotone and convex constraints*
Lengning Liu and Miroslaw Truszczynski
In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, July, 2005, Pittsburgh, Pennsylvania, U.S.A. Page 701 - 706, AAAI Press

iii. *Local search with bootstrapping*

Lengning Liu and Miroslaw Truszczynski

In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, May 2004. Vancouver, Canada

iv. *WSAT(CC) - a fast local-search ASP solver*

Lengning Liu and Miroslaw Truszczynski

In *Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning*, January 2004. Fort Lauderdale, Florida, U.S.A. Page 351 - 355, LNCS 2923, Springer

v. *Local-search techniques for propositional logic with cardinality constraints*

Lengning Liu and Miroslaw Truszczynski

In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, September, 2003. Kinsale, County Cork, Ireland. Page 495 - 509, LNCS 2833, Springer

vi. *Children Turing Test*

Ruqian Lu, Songmao Zhang, Lengning Liu, Fan Yang, Xiaolong Jin, Chong Zhao, Hong Zheng, Hongge Liu, Mo Wang

In *Proceedings of the Second China-Japan Natural Language Processing*, October 30 - November 2, 2002. Peking University, China

vii. *Talk to Computer in Natural Style*

Ruqian Lu, Songmao Zhang and Lengning Liu

In *Proceedings of the International Symposium on Future Software Technology*, October 27 - 29, 1999. Nanjing, China

(c) Papers Accepted

i. *Properties and applications of monotone and convex constraint programs*

Lengning Liu and Miroslaw Truszczynski

Accepted by JAIR, pending for revision

ii. *Local search techniques for boolean combinations of pseudo-boolean con-*

straints

Lengning Liu and Miroslaw Truszczynski

To appear in Proceedings of AAAI-06, July 16 - 21, 2006, Boston, USA

(d) Master's Dissertation

i. Children Turing Test (in Chinese)

Lengning Liu

Institute of Mathematics, Chinese Academy of Science, Beijing, China, 2000

(e) Technical Report

i. *Aspps Users Manual*

Deborah East, Lengning Liu, Stephen Logston, Victor Marek and Miroslaw Truszczynski

Department of Computer Science, University of Kentucky, July, 2004, Lexington, Kentucky, U.S.A.