2011

ON SIMPLE BUT HARD RANDOM INSTANCES OF PROPOSITIONAL THEORIES AND LOGIC PROGRAMS

Gayathri Namasivayam
University of Kentucky, gayatrina@gmail.com

ABSTRACT OF DISSERTATION

Gayathri Namasivayam

The Graduate School

University of Kentucky

2011

ON SIMPLE BUT HARD RANDOM INSTANCES OF PROPOSITIONAL THEORIES AND LOGIC PROGRAMS

ABSTRACT OF DISSERTATION

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

By

Gayathri Namasivayam

Lexington, Kentucky

Director: Dr. Mirosław Truszczyński, Department of Computer Science

Lexington, Kentucky

2011

ABSTRACT OF DISSERTATION

ON SIMPLE BUT HARD RANDOM INSTANCES OF PROPOSITIONAL THEORIES AND LOGIC PROGRAMS

In the last decade, Answer Set Programming (ASP) and Satisfiability (SAT) have been used to solve combinatorial search problems and practical applications in which they arise. In each of these formalisms, a tool called a solver is used to solve problems. A solver takes as input a specification of the problem – a logic program in the case of ASP, and a CNF theory for SAT – and produces as output a solution to the problem. Designing fast solvers is important for the success of this general-purpose approach to solving search problems. Classes of instances that pose challenges to solvers can help in this task.

In this dissertation we create challenging yet simple benchmarks for existing solvers in ASP and SAT. We do so by providing models of simple logic programs as well as models of simple CNF theories. We then randomly generate logic programs as well as CNF theories from these models. Our experimental results show that computing answer sets of random logic programs as well as models of random CNF theories with carefully chosen parameters is hard for existing solvers.

We generate random logic programs with 2-literals, and our experiments show that it is hard for ASP solvers to obtain answer sets of purely negative and constraint-free programs, indicating the importance of these programs in the development of ASP solvers. An easy-hard-easy pattern emerges as we compute the average number of choice points generated by ASP solvers on randomly generated 2-literal programs with an increasing number of rules. We provide an explanation for the emergence of this pattern in these programs. We also theoretically study the probability of existence of an answer set for sparse and dense 2-literal programs.

We consider simple classes of mixed Horn formulas with purely positive 2- literal clauses and purely negated Horn clauses. First we consider a class of mixed Horn formulas wherein each formula has m 2-literal clauses and k-literal negated Horn clauses. We show that formulas that are generated from the phase transition region of this class are hard for complete SAT solvers. The second class of Mixed Horn Formulas we consider are obtained from completion of a certain class of random logic programs. We show the appearance of an easy-hard-easy pattern as we generate formulas from this class with increasing numbers of clauses, and that the formulas generated in the hard region can be used as benchmarks for testing incomplete SAT solvers.

KEYWORDS: Knowledge representation, Answer-set programming, Propositional satisfiability, Mixed Horn formulas, Random SAT

Gayathri Namasivayam

Student's signature

February 15, 2011

Date

ON SIMPLE BUT HARD RANDOM INSTANCES OF PROPOSITIONAL THEORIES AND LOGIC PROGRAMS

By

Gayathri Namasivayam

Dr. Mirosław Truszczyński
Director of Dissertation

Dr. Raphael Finkel
Director of Graduate Studies

February 15, 2011

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

Gayathri Namasivayam

The Graduate School

University of Kentucky

2011

ON SIMPLE BUT HARD RANDOM INSTANCES OF PROPOSITIONAL THEORIES AND LOGIC PROGRAMS

DISSERTATION

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

By

Gayathri Namasivayam

Lexington, Kentucky

Director: Dr. Mirosław Truszczyński, Department of Computer Science

Lexington, Kentucky

2011

DEDICATION

To my parents

Sakuntala Shakher and Namasivayam Selvarajan

ACKNOWLEDGMENTS

I was fascinated by the ability to represent the knowledge about a real world situation in a logical language, as well as the ability to use logic to reason from this knowledge. I had the dream to learn more about this area, and I came to the University of Kentucky for further studies unaware that I would be able to make this wish come true. I met my advisor, Dr. Mirosław Truszczyński, in the computer science department and was extremely thrilled that I could pursue further studies in this area of my dreams under his valued guidance. Dr. Truszczyński provided me with ongoing knowledge, guidance, and support from the first until this moment. My first memory is of Dr.Truszczyński teaching me the logical language PS+ and helping me use it to represent several combinatorial problems. I am so thankful and extremely grateful to Dr. Truszczyński for teaching me the preliminaries of logics and completely guiding my research, as well as for giving me the opportunity to meet several researchers in my field.

I am thankful to Dr. Victor Marek, Dr. Judy Goldsmith, and Dr. Jerzy Jaromczyk for teaching and guiding me through several classes related to my area of study. They have always motivated and guided me during my study at this department.

I thank Dr. Raphael Finkel and Dr. Grzegorz Wasilkowski for their prompt help and guidance during my graduate study at the department.

I am thankful to Dr. Victor Marek, Dr. Judy Goldsmith, Dr. Kevin Donohue, and Dr. Uwe Nagel for being on my committee and reading drafts of my dissertation.

I would like to thank Lengning Liu, Liangrong Yi, Krol Kevin Mathias, Peng Dai, Nicholas Mattei, and Joshua Guerin for all the research related discussions and the good times they shared with me at our AI lab.

I am so happy to have made wonderful friends Pete Wilson, Tanya Floyd, Thomas Goodness, Karen Gerstandt, Ramakanth Kavuluru, Bev McChesney, and Jim McChesney during my study at the University of Kentucky, and I thank them for all the affection they showered upon me and for being there for me at all times.

I thank my husband Sivamoorthy Shanmugam for being supportive of my stay and study away from him.

I thank my parents for their love. They have continuously supported and motivated me to pursue my doctorate degree. They have been the pillar of support for me at all times during my study.

Lastly, I thank my sister Gangothri Namasivayam for visiting me, and my grandmother Kalaimagal Shakher for traveling several times to Lexington to visit me and be with me during my study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The past few decades have seen the development of many approaches to solving search problems. We consider here two of them: reducing the problem to Satisfiability (SAT), and to Answer Set Programming (ASP).

Satisfiability, or SAT for short, is a problem arising in propositional logic. We define it formally later. Here we simply mention that it consists of deciding whether a propositional formula (often subject to some syntactic restrictions) has a satisfying assignment (or *model*). Many real-world problems can be reduced to SAT in a way that establishes a correspondence between solutions to the problem and models of the corresponding formula. To explain more precisely what we have in mind, let us consider a problem Π with the set of instances $\mathcal{I}(\Pi)$. By the propositional logic formulation of Π we mean a function f_Π, such that

1. to every instance $I \in \mathcal{I}(\Pi)$, f_Π assigns a propositional formula $f_\Pi(I)$
2. f_Π can be computed in polynomial time
3. for every instance $I \in \mathcal{I}(\Pi)$, Π has a solution for an instance I if and only if $f_\Pi(I)$ has a model
4. solutions to Π for I can be extracted in polynomial time from models of $f_\Pi(I)$

It is known that the class of problems that can be reduced in this way to SAT is the class NP-search also known as the class NPMV [67]. It includes many hard problems arising in practical applications.

Answer Set Programming (ASP) [53, 61] is a knowledge representation and reasoning formalism that is used to model and solve computationally hard search problems. In ASP an instance of the search problem is modeled as a logic program so that *answer sets* of

the logic program correspond to solutions of the problem. ASP provides a programming environment for modeling the constraints and the domain information (an instance) of the search problem as a high-level program using an ASP language [70, 72]. ASP uses a tool called the *grounder* to convert the high-level program and domain information into a low-level program called a *propositional* program. In more formal terms, let Π be a problem with the set of instances $\mathcal{I}(\Pi)$. The ASP functions f_Π, similar to those in SAT, are determined by a fixed-length ASP program P_Π capturing the specification of the problem Π and by the process called grounding denoted by $grnd$. More precisely, if the program P_Π encodes Π and I represents an instance to Π, then $f_\Pi(I) = grnd(P_\Pi \cup I)$. The programming environment, along with the grounder, constitutes the front-end of ASP. The back-end of ASP has a *solver* that takes the ground propositional program produced by the front-end and computes its answer sets.

The primary difference between the two formalisms is that ASP comes with a high-level programming front end that is not provided by SAT. A single high-level programming front end can take different problems and their instances and produce corresponding propositional programs that are then solved by the solver. In the case of SAT, each problem and its instance must be reduced to a propositional formula to be solved by a SAT solver using specialized tools that differ from problem to problem. However, the solvers for the two formalisms employ similar techniques for solving propositional programs and formulas.

The two formalisms have a wide range of applications [27, 48]. The current ASP languages are used to model instances of real-world problems such as preference reasoning [14, 15], semantic web [40], product configuration [71], software configuration [75], and combinatorial problems in the class NP and in the class Σ_2^P. The main applications of SAT are in the areas of hardware verification [18], bounded model checking [17], software verification [21], and planning [44].

Applications of SAT and ASP created the need for good solvers that can solve problems quickly. Hence many solvers have been developed in recent years for each of these formalisms, and have been successful in solving search problems and problems of practical importance. There are two broad classes of solvers: *complete* solvers are those that always

find a solution to a problem or determine that none exists; *incomplete* solvers are those that do not guarantee to find a solution even if one exists. The solvers in each of these classes can be further characterized based on the algorithms, data structures, and heuristics used within them.

In order to compare the performance of these solvers, several benchmark problems were proposed for both SAT and ASP at the SAT competitions [2], ASP competition [3, 5], and at the online benchmarking environment for ASP [1]. Some of these benchmarks include:

1. Real-world problems: routing, job-shop scheduling, configuration problems, traveling salesperson, grammar-based information extraction, formal verification of processors, and bounded model checking.

2. Graph problems: weighted spanning tree, weighted bounded dominating set, Hamilton cycle, Hamilton path, reachability, hierarchical clustering, connected dominating set, graph partitioning, and graph coloring.

3. Puzzles/Games: 15-puzzle, sudoku, solitaire, towers of Hanoi, maze generation, blocked N-queens, and sokoban.

4. Random: random tight and non-tight logic programs, and random k-SAT formulas.

Even though many challenging benchmark problems have been designed and generated to compare the performance of SAT solvers, the generation of random SAT benchmarks in Conjunctive Normal Form (CNF) has been of special interest to the researchers in the SAT community. Researchers have proposed different models of random CNF theories. In particular, a *fixed clause length model* [57] with a carefully selected set of parameters was shown to generate hard CNF theories for SAT solvers. These benchmarks led to significant advances in building efficient SAT solvers, both complete and incomplete.

The focus of this thesis is on generating random logic programs and random CNF theories that can be shown to be hard for solvers based on a measure such as the time taken by the solver to obtain a solution, or using choice points as a parameter that measures the size of the search space traversed by the solver to obtain a solution.

Our work is related to the prior work done on generating random CNF formulas [57], as well as to the work done by Zhao and Lin on generating random logic programs [80]. We provide a description of the previous work in Chapter 5. Our work, like the prior work done on generating random logic programs and random CNF theories, focuses on creating hard instances for solvers. The main difference between the prior work and ours is that we focus on generating *simple* yet hard random programs and theories.

1.1 Motivation

Our study and generation of random theories has been motivated by two key considerations. First, hard random theories can be used as benchmarks to evaluate the algorithms used in existing solvers. Second, experimental analysis as well as theoretical studies of the properties of these random theories can provide us with insights for improving the design of the heuristics and algorithms in these solvers.

1.2 Main Contributions

1. We provide models of simple logic programs that consist of 2-literal rules. These programs can contain constraints (i.e., rules of the form $\leftarrow a, b$; $\leftarrow a, not(b)$ and $\leftarrow not(a), not(b)$), purely negated rules of the form $a \leftarrow not(b)$, and purely positive rules of the form $a \leftarrow b$. We define different classes of logic programs that contain combinations of these rules, such as logic programs that are purely negative (i.e., contain only negated rules) without constraints, or those that contain both positive as well as purely negative rules.

   We also provide models of simple SAT formulas, which are special types of Mixed Horn Formulas (MHFs) that predominantly consist of purely positive 2-literal clauses and purely negative Horn clauses. We also show that the SAT problem for this class of CNF formulas is NP-complete.

2. We randomly generate 2-literal logic programs and SAT formulas from these models and study their properties. In the case of randomly generated logic programs, we

experimentally and theoretically study the probability distribution for the existence of an answer set for the 2-literal programs that we generate with an increasing density of rules. We also experimentally study the difficulty for ASP solvers of the different classes of randomly generated 2-literal logic programs and show that the programs that have rules that are purely negative and constraint-free form the hardest class. In addition, we observe the existence of an easy-hard-easy pattern as we compute the average number of choice points generated by a solver on computing an answer set for this class of logic programs, and we provide arguments that help explain the occurrence of this pattern.

We experimentally study the probability of the existence of a satisfying truth assignment for randomly generated MHFs. We observe a phase-transition for the probability of the existence of a model and a corresponding easy-hard-easy pattern for SAT solvers, as we test the satisfiability of MHFs, that we generate with an increasing number of 2-literal clauses and a fixed number of Horn clauses.

3. We generate hard benchmarks for logic program solvers and SAT solvers. We show that computing answer sets, for random 2-literal logic programs that have purely negative constraint-free rules, are hard for ASP solvers.

   We show that MHFs that are generated from the critical region where the probability of the randomly generated formula being satisfiable is 0.5 are hard for SAT solvers. We also generate hard random MHFs for incomplete solvers.

1.3 Thesis Organization

We provide in the subsequent chapter a brief introduction to the two formalisms Answer Set Programming and Satisfiability. In Chapter 3 we provide a model for the generation of simple-yet-hard random logic programs and study the properties of these programs. We then consider models for the random generation of simple classes of mixed Horn formulas in Chapter 4, and show that these formulas can be used as hard benchmark problems for existing SAT solvers. We provide in Chapter 5 a brief discussion of related works. Finally,

we conclude in Chapter 6 and provide future research directions.

Chapter 2

The two Formalisms

2.1 Satisfiability of Propositional Theories

This section explains the syntax and semantics of propositional logic needed to define the satisfiability (SAT) problem. A description of a few subclasses of SAT such as Horn, Mixed Horn Formulas (MHF), and K-CNF formulas is given here as well. We also provide a brief introduction to SAT solvers.

2.1.1 Syntax and Semantics

In the area of SAT, a problem is usually represented as a special type of formula in propositional logic called a Conjunctive Normal Form (CNF) formula, which is formed from boolean variables using boolean connectives $\wedge$, $\vee$, $\neg$, and parentheses. We often write simply *variables* instead of boolean variables. The syntax and semantics of CNF formulas is given below.

Let v denote a boolean variable. A literal is either a boolean variable v or its negation $\neg v$. A *clause* C is a disjunction $l_1 \vee l_2 \vee \ldots l_m$, where all l_i's are literals. A CNF formula is a conjunction of clauses.

We consider two truth values: **t** (true) or **f** (false). An *assignment* is a function that maps boolean variables to truth values. Let F be a CNF formula and A an assignment that maps every boolean variable in F to a truth value. The truth value of F based on A is then computed inductively as follows. The truth value of a boolean variable that is mapped to true in A is true, otherwise it is false. The negation of a variable is true if the variable is assigned false in A, and false otherwise. A clause is evaluated as true in A (i.e., its truth value is true in A) if at least one of its literals is true in A. A CNF formula is evaluated to be true in A if all of its clauses are true in A.

A CNF formula is *satisfiable* (consistent) if there is at least one assignment that satisfies it. Otherwise, the formula is *unsatisfiable* (inconsistent). Each assignment satisfying a

formula is called a satisfying assignment or a model. An empty clause has no satisfying assignments, and the empty conjunction of clauses is satisfied by every assignment. With these definitions we will now define the satisfiability problem in its search version.

Definition 1. *By the satisfiability (SAT) problem we mean the problem in which inputs are formulas in the CNF form and the objective is to find one or more satisfying assignments, or determine that no such assignment exists.*

The decision version of the SAT problem consists of deciding for an input CNF formula F whether it has a model (without the requirement that if such a model exists then it has to be returned).

2.1.2 Examples

Every problem in the class NP-search [67] can be reduced to SAT as described in the introduction. We provide here the SAT representation for two problems: graph coloring and blocked n-queens.

Graph coloring

In the graph coloring problem denoted by π_{col}, we are given a set of colors and a graph. The goal of the problem is to color every vertex in the graph with a color so that no two vertices that are connected by an edge have the same color. Every instance I of the graph coloring problem is given by a set C of k colors and a graph G with a set $V = \{v_1, \ldots, v_n\}$ of n vertices and set of edges E. We construct here the propositional formula (i.e., a set of clauses) $f_{col}(I)$ for an instance of the graph coloring problem in the following way.

Let b_{ij} be a boolean variable representing the statement that vertex $v_i \in V$, $1 \leq i \leq n$ is colored with color $c_j \in C$, $1 \leq j \leq k$. First, we include in $f_{col}(I)$ the clauses

$$b_{i1} \vee \ldots \vee b_{ik},$$

for every i, $1 \leq i \leq n$. This ensures that each vertex is associated with at least one color. Next, we include in the set $f_{col}(I)$ clauses of the form

$$\neg b_{ij} \vee \neg b_{ij'}$$

for every i, j and j' where $1 \leq i \leq n$, and $1 \leq j, j' \leq k$. This group of clauses ensures that each vertex is associated with at most one color. Let $\{v_i, v_i'\} \in E$ represent an edge between a pair of vertices v_i and v_i'. To ensure that no two vertices of an edge are colored using the same color, we include in $f_{col}(I)$ clauses of the form

$$\neg b_{ij} \vee \neg b_{i'j},$$

where $i < i'$ and $\{v_i, v_i'\}$ is an edge in the graph.

Theorem 1. *Let I be an instance to the graph coloring problem. Then solutions to I are in a one-to-one correspondence with models of the propositional formula $f_{col}(I)$.*

Proof. Let M be a model of $f_{col}(I)$. Then we construct a solution S_M for an instance I of the graph coloring problem in the following way: for every boolean variable b_{ij} that is assigned true in M, we color the vertex v_i with color c_j in S_M.

First, we show here that if M is a model of the propositional theory $f_{col}(I)$, then S_M is a solution of the coloring problem for I. Let us assume that M is a model of $f_{col}(I)$. One can check the following:

- In S_M, each vertex v_i is colored with at least one color c_j (the first group of clauses).
- In S_M, each vertex v_i is be colored with at most one color c_j (the second group of clauses).
- In S_M, any two vertices that are connected by an edge cannot be colored using the same color (the third group of clauses).

Hence, it follows that in S_M every vertex is colored with a single color and that any two vertices that are connected by an edge cannot be colored using the same color. Hence S_M is a solution to the graph coloring instance I.

Conversely, we show that if S is a solution to an instance I, then we construct a model M_S from S in the following way: for every vertex v_i that is colored with a color j, we assign a boolean variable b_{ij} to be true in M_S; otherwise, we set b_{ij} to false in M_S. Let S be a solution to a graph coloring problem. Since in S every vertex v_i in the graph is colored

with a single color c_j, we know that all the clauses in the first two groups are true in M_S. Moreover, since no edges have both their vertices assigned the same color, every clause in the third group is true in M_S, too. Thus, M_S is a model of $f_{col}(I)$.

□

Blocked n-queens

The blocked n-queens problem is a variant of the n-queens problem. In the blocked n-queens problem, we have an $n \times n$ board and n queens. Each square on the board is formed by the intersection of a particular row and column. A square can hold at most one queen. Some squares are blocked. The goal of the problem is to place n queens on the unblocked squares of the board so that no two queens are placed on the same row, column, or diagonal. Let us use i, where $1 \leq i \leq n$, to denote the ith row on the board and j, where $1 \leq j \leq n$, to denote the jth column.

We construct here the propositional formula for an instance of the blocked n-queens problem in the following way. We will define a set of clauses $f_{bq}(I)$ representing this instance of the blocked n-queens problem. Let q_{ij} be a boolean variable representing the statement that there is a queen in the square formed by the ith row and jth column. Some squares on the board are blocked and cannot hold any queen. For every blocked square (i, j) include in the set of clauses $f_{bq}(I)$,

$$\neg q_{ij}. \tag{2.1}$$

At least one queen must be placed on every column. This constraint is represented by clauses of the form

$$q_{1j} \vee q_{2j} \vee \ldots \vee q_{nj},\ j = 1, 2, \ldots, n. \tag{2.2}$$

A conflict arises when any two queens are assigned to the same column, row, or diagonal. These constraints are represented by clauses of the forms

$$\neg q_{ij} \vee \neg q_{i'j}, \tag{2.3}$$

where $i \neq i'$,

$$\neg q_{ij} \vee \neg q_{ij'}, \tag{2.4}$$

where $j \neq j'$, and

$$\neg q_{ij} \vee \neg q_{i'j'} \tag{2.5}$$

where the absolute value of $i - i'$ is equal to the absolute value of $j - j'$, $i \neq i'$, and $j \neq j'$.

Theorem 2. *Let I be an instance of the blocked n-queens problem. Then, solutions to the problem for I are in a one-to-one correspondence with models of the propositional theory $f_{bq}(I)$.*

Proof. Let M be a model of $f_{bq}(I)$. Then we construct a solution S_M of I in the following way: for every boolean variable q_{ij} that is assigned to true in M, we place a queen in the square (i, j). One can then check that in S_M no queen is placed in a blocked square (since M satisfies clauses of type (2.1)). Moreover, there is a queen in every column (clauses of type (2.2)). Next, in S_M no two queens are contained in a column (clauses of type (2.3)). In particular, it follows that exactly n queens are placed on the board. Next, no two queens are contained in the same row (clauses of type (2.4)) and no two queens are placed on the same diagonal (clauses of type (2.5)). Thus, S_M is a solution for I.

Conversely, let S be a solution of an instance I of the n-queens problem. We construct from S a model M_S in the following way: for every queen that is placed in square (i, j), we assign the boolean variable q_{ij} to true in M_S. One can check that M_S satisfies all of the clauses in $f_{bq}(I)$. Hence, M_S is a model of $f_{bq}(I)$. For instance, since in S no two queens appear in the same column, all clauses of type (2.3) are true in M_S.

□

2.1.3 Special classes of SAT formulas

In addition to a general class of CNF formulas, we consider in the thesis other special subclasses of CNF formulas with a restricted syntax. A CNF formula in which every clause has exactly k literals is called a *k-CNF* formula. A *Horn* clause is a clause with at most one occurrence of a non-negated atom. A 2-literal clause is a clause with exactly 2-literals in it. We define *Horn* to be the class of CNF formulas in which every clause is a Horn clause.

Mixed Horn Formulas (MHFs) are a subclass of CNF formulas in which every clause in the formula is either a 2-literal clause or a Horn clause.

The general interest within the SAT community for consideration of each of these different restricted classes of formulas is due to some of the following reasons: simplicity in the structure of the formulas as in the case of 2-CNF, Horn, and mixed Horn formulas; the ability to represent several real-world applications within the restricted class of MHFs; and a computational advantage allowing 2-CNF and Horn formulas to be solved in polynomial time.

2.1.4 Complexity

The problem of deciding if a CNF formula is satisfiable is NP-complete, and is the first problem shown to be NP-complete [19]. The satisfiability of 2-CNF, as well as Horn formulas, can be solved in polynomial time [9, 56], and the satisfiability of k-CNF formulas with $k \geq 3$ is NP-complete [19]. However, the satisfiability of mixed Horn formulas, interestingly enough, is also NP-complete [64].

Schaefer's dichotomy theorem distinguishes classes of instances of the boolean constraint satisfaction problem for which a solution can be found in polynomial time [66]. The classes of instances for which a polynomial time algorithm exists are: 2-SAT, Horn, dual-Horn, trivially satisfiable formulas (i.e, a class of CNF formulas in which every clause has at least a single non-negated variable or a class of CNF formulas in which every clause has at least a single negated variable), and affine formulas. Schaefer also showed that all other classes of the boolean constraint satisfaction problem defined in terms of constraint types are NP-complete.

2.1.5 SAT solvers

A SAT solver is a program that takes a SAT instance as input and produces as output a model, or all models, or decides that there are no models for it (i.e., it is unsatisfiable). SAT solvers can be broadly classified into two categories: complete and incomplete solvers. Complete solvers are those that are guaranteed to always find a solution for a SAT instance

if one exists, or to determine that the instance is unsatisfiable by searching through the space of truth assignments. Incomplete solvers are those solvers that are not guaranteed to find a solution of a SAT instance, even if one exists (but they often do, and do so fast), and do not determine that the instance is unsatisfiable.

Many complete SAT solvers have been developed, and the more recent ones that were the winners at the SAT competition in 2009 are *glucose* [10], *satzilla* [77], and *march_hi* [39]. The earliest procedure used in complete SAT solvers is the Davis-Putnam-Logemann-Loveland (DPLL) procedure. The procedure shown in Figure 2.1 is based on the DPLL algorithm given in [12]. The DPLL procedure takes as input a CNF formula and either produces as output a satisfying truth assignment or determines that the formula is unsatisfiable [22]. This procedure systematically searches through all possible boolean assignments until it finds a model or shows that the given formula is unsatisfiable. It does so by a search process that creates a binary tree whose nodes represent variables in the tree, and the two downward edges from a node in the tree correspond to the two possible variable assignments made to the variable at the node. Once a variable v in the formula F is assigned a value Val (i.e., $Val \in \{true, false\}$), then the DPLL procedure eliminates from F the variable v and obtains a reduced formula $F|(v = Val)$ in the following way: if there is a clause with a literal (either v or $\neg v$) that evaluates to false, then it eliminates that literal from the clause; and if there is a clause that contains a literal (either v or $\neg v$) that evaluates to true, then that clause is satisfied by the current partial truth assignment made to the variable and it is removed.

Example 3. *Consider a CNF formula F with variables $\{v_1, v_2, v_3, v_4\}$, and*

$$F = \{v_1 \vee \neg v_2,$$

$$v_3 \vee \neg v_4,$$

$$\neg v_1 \vee v_2\}.$$

Then if v_2 is assigned $false$. The reduced formula determined by the DPLL procedure is

$$F|(v_2 = false) := \{v_3 \vee \neg v_4,$$

$\neg v_1\}$.

$\triangle$

The following are the steps performed by a DPLL search algorithm as described in Figure 2.1.

1. The procedure initially performs a step called *unit propagation* as seen in Line 1. The unit propagation algorithm takes as input a CNF formula I and assigns it to F. It then determines clauses called *unit clauses* that has a single unassigned literal. The algorithm on detection of a unit clause immediately assigns the variable v in the single unassigned literal in the unit clause to a boolean value Val that forces the unassigned literal now to evaluate to true. The unit propagation procedure then eliminates the variable v from the formula F (as described above) and obtains a reduced formula $F|(v = Val)$, which it calls F. The unit propagation procedure continues to look for other unit clauses and eliminates them. During unit propagation, more and more unit clauses can appear as more and more variables are assigned with values. The formulas gets reduced each time a unit clause is obtained. This unit propagation step terminates when either it can no longer detect any new unit literal clauses, or when a clause has been falsified (i.e., one of the clauses is an empty clause $\{\}$). The algorithm outputs a partial assignment A, and a reduced CNF formula F.

2. The procedure then checks to determine if all the variables are assigned; if so then the formula is satisfiable and the procedure terminates with a satisfying assignment (see Lines 2–4). The procedure reports the formula to be unsatisfiable if unit propagation discovered a conflict (i.e., the reduced formula returned by unit propagation has an empty clause $\{\}$, see Line 5–6).

   If there are unassigned variables then it starts by choosing one variable v among them, assigning it to be either *true* or *false* and performing unit propagation (see Line 8). The DPLL procedure provided above is a recursive procedure; hence the call to the unit propagation algorithm is made by calling the DPLL procedure with the

INPUT: I: a CNF instance of SAT
OUTPUT: A: a satisfying assignment for I output
SOL: indicates if the instance is satisfiable or unsatisfiable

BEGIN

1. (A, F) = Unit-Propagation(I)
2. **If** all variables in F are assigned **then**
3. $SOL =$ Satisfiable;
4. **return** A;
5. **Else If** a clause in F is falsified **then**
6. $SOL =$ Unsatisfiable;
7. **return** false;
8. **Else** choose a variable v in F
9. **If** $(V = (DPLL(F|v = true)) == false$ **then**
10. **If** $(V = (DPLL(F|v = false)) == false$ **then**
11. $SOL =$ Unsatisfiable;
12. **return** false;
13. **Else**
14. **return** $(V \cup A \cup \{v = false\})$
15. **Else**
16. **return** $(V \cup A \cup \{v = true\})$

END

Figure 2.1: Algorithm $DPLL$

reduced formula $F|(v = true)$ or $F|(v = false)$. Each time a variable is chosen, a node is added in the search tree and this node is called a *choice point*. The assignment of a value to a variable (i.e., $v = true$ or $v = false$) is represented by a branch in the tree from the corresponding node.

3. Based on the current partial assignment made to the variables in the formula (i.e., after unit propagation), DPLL checks to see if any clause in the formula has been falsified. If none of the clauses have been falsified, the algorithm repeats steps 2–4; otherwise if a clause is falsified, then it does one of the following steps.

   - The procedure backtracks to a choice point in the search tree that has most recently been assigned exactly one of the two boolean values and then reassigns it to the opposite value, performs unit propagation, and checks again to see if a clause is not satisfied (see Line 10).
   - If there is no earlier choice point for the algorithm to backtrack to in the previous step, the procedure determines that the formula is unsatisfiable.

The algorithms used in more recent complete SAT solvers are variants and modification of the DPLL procedure. Some of the additional successful procedures that were integrated with the DPLL framework include *lookahead* [30], backjumping [11], and *conflict-driven clause learning* [73].

Incomplete solvers, otherwise called *stochastic local search* (SLS) solvers, have been successful in obtaining models of satisfiable formulas with a large number of variables in a shorter time than complete solvers, especially for the class of hard randomly generated formulas. They are predominantly run with a fixed time limit within which they try to find a model of the formula. These solvers do not necessarily guarantee finding a model within the time given to them. The recent high-performing SLS solvers are *TNM* [4], *gNovelty+* [4], and hybridGM [4].

The earliest SLS algorithm is *GSAT* [69]. It performs a randomized local search by initially generating a random complete assignment. The algorithm then reassigns (i.e., *flips*) a single variable to the opposite value in the assignment. The variable that is chosen

to be flipped is the one that minimizes the number of unsatisfied clauses in the formula. The solver greedily flips variables until it finds a satisfying assignment or until a predetermined number of maximum flips is reached. Once the maximum number of flips has been reached, the SLS repeats the entire process by generating, once again, a random complete assignment. The SLS repeats the entire process until the time limit is reached.

GSAT has the possibility of getting stuck for a long time in a *local minima* which is where all the neighboring truth assignments (those that can be reached by a flip) do not result in decreasing the number of unsatisfied clauses. *GSAT* could spend considerable time in such a local minima. Hence, in order to improve on such a situation, *random walk* was introduced within local search solvers. The random walk strategy occasionally allows a random selection of a variable to flip as opposed to the greedy flip strategy described earlier. The random walk is incorporated within the *walksat* solver [68], in addition to the greedy flip strategy that was used in *GSAT*. *Walksat* chooses a falsified clause and performs the following: If there exists a variable in the chosen falsified clause that on flipping does not make any satisfied clause falsified, it flips the variable; otherwise, it does a random walk by randomly choosing a variable to flip from the chosen falsified clause with probability p and by greedily choosing a variable to flip also from within a falsified clause with probability $1 - p$. The random walk strategy has proven to be successful on larger random 3-SAT instances when compared to the greedy strategy used in *GSAT* [69].

2.2 Answer Set Programming

Answer Set Programming (ASP for short) evolved from logic programming as the result of the research on the meaning of negation in the syntax of logic programming. The semantics that gained overwhelming acceptance, the *stable-model* semantics, could not be reconciled with the single-intended model paradigm of logic programming. As examples that we give later show, it is quite common for a program to have multiple stable models. Thus, in order to exploit the stable-model semantics for logic programming a shift in the paradigm was needed. Such a shift was proposed in the late 1990s [53, 61]. Under the new paradigm, stable models represent objects to compute, and by intention, each represents a solution

to the problem modeled by the program. Stable models are otherwise called as *answer sets* and hence the name Answer Set Programming. Since then ASP has become one of the most vibrant areas of research in logic programming. One of the important factors behind the phenomenon of ASP is its strong connection to knowledge representation and non-monotonic logics, in particular to default logic by Reiter [65]. More specifically, logic programs with the stable-model semantics can be viewed (in a quite direct way) as special default theories.

We provide here the syntax and the semantics of ASP. We also discuss other concepts that are relevant to the thesis, such as: completion semantics, positive dependency graph, supported model, and loop formulas. A brief overview of the complexity analysis of normal logic programs is given. We end the chapter with a discussion of the different kinds of ASP solvers.

2.2.1 Syntax

Let A be a nonempty set of symbols. Each symbol a, where $a \in A$, is called an *atom*. A *normal rule* is an expression of the form

$$a \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m),$$

and a *constraint* is a an expression of the form

$$\leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m),$$

where a, b_i's and c_j's are atoms and not is the *negation as failure* connective. By $head(r)$ we represent the atom a, also called the *head* of the normal rule r. We use the term rule to represent either a normal rule or a constraint. We denote the set of negated atoms c_j's (the set of non-negated atoms b_i's) in a rule r as $negbody(r)$ $(posbody(r))$. A normal rule r with the empty $posbody(r)$ as well the empty $negbody(r)$ is a *fact*, and it is usually represented with an omitted rule connective as

$$a.$$

A *normal logic program* P is composed of rules. By $At(P)$ we denote the set of all atoms in the program P.

Example 4. *Example of a normal logic program P with $At(P) = \{a, b, c, d\}$ is shown here.*

$$\{a \leftarrow not(b).$$

$$b \leftarrow c, not(d).$$

$$c.\}$$

△

Let M be a set of atoms where $M \subseteq At(P)$. We define the satisfaction relation $\models$ as follows.

- $M \models a$ if $a \in M$.
- $M \models not(a)$ if $a \notin M$.
- For a constraint r, $M \models r$ if there exist an atom a, such that.
  $a \in posbody(r)$ and $M \not\models a$, or
  $a \in negbody(r)$ and $M \not\models not(a)$.
- For a rule r that is not a constraint, $M \models r$ if $head(r) \in M$ whenever
  for every atom $a \in posbody(r)$, $M \models a$, and
  for every atom $a \in negbody(r)$, $M \models not(a)$.
- M is a *model* of a normal logic program P, denoted by $M \models P$, if for every rule $r \in P$, $M \models r$.

A normal logic program whose rules have no negated literals in their bodies as well as no constraints is called a *Horn logic program*. A Horn logic program always has a unique least model [26]. We use $lm(P)$ to denote the least model of a Horn logic program P. We provide here an example of a Horn logic program.

Example 5. *Let P be a Horn logic program.*

$$P = \{a \leftarrow b.$$

$$b \leftarrow c, d.$$

$$b.\}$$

Its unique least model is $\{a, b\}$. △

Normal logic programs that are not Horn may or may not have a least model. An example of such a normal logic program that does not have a least model is shown in Example 6.

Example 6. *Let* P *be a normal logic program:*

$$\{a \leftarrow not(b).$$

$$b \leftarrow not(a).\}$$

Then P *has three models* $\{a\}$, $\{b\}$, *and* $\{a, b\}$. *Thus,* P *does not have a least model. It has two minimal models* $\{a\}$ *and* $\{b\}$. △

2.2.2 Stable-model semantics of a normal logic program

The semantics of a normal logic program is based on the intuition that if positive atoms in the body of the rule are true and the negated atoms in the body can not be proved then the atom in the head of the rule must be true.

In this subsection, we present the stable-model semantics of a propositional normal logic program [34]. Let M be a model of P. We define here the reduct P^M of a normal logic program P. The reduct P^M of a normal logic program P with respect to a model M is computed as follows. For each rule $r \in P$, where $r = a \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m)$,

1. if $M \cap \{c_1, \ldots, c_m\} \neq \emptyset$, delete r, and

2. if r has not been deleted, remove all negated atoms $(not(c_1), \ldots, not(c_m))$ from r.

The logic program P^M does not contain any occurrences of not connectives. Therefore, it is a Horn logic program. A set of atoms $M \subseteq At(P)$ is an *answer set* if M is the least model of P^M (i.e., $M = lm(P^M)$).

We provide an example of a normal logic program that has answer sets in Example 7, and also an example of one that does not have an answer set in Example 8. We also note that if M is an answer set of P then M is a model of P [34]. Hence, in these examples we will compute answer sets of P by considering only subsets of atoms of P that are models of P.

Example 7. *Consider a normal logic program P with rules*

$$\{a \leftarrow not(b).$$

$$b \leftarrow c,\ not(a).$$

$$c.\}$$

The program P has three models $M_1 = \{a, c\}$, $M_2 = \{b, c\}$ and $M_3 = \{a, b, c\}$. The reduct of P w.r.t. the model M_3, P^{M_3} is given by

$$P^{M_3} = \{c\}.$$

The unique least model of P^{M_3} is $\{c\}$ which is different from M_3. Thus, M_3 is not an answer set. We can show that M_1 and M_2 are answer sets in the following way. The reduct of P w.r.t. the model M_1, P^{M_1} is

$$P^{M_1} = \{a, c\}$$

Clearly, $lm(P^{M_1}) = \{a, c\}$. Since, $lm(P^{M_1}) = M_1$, M_1 is an answer set of P. The reduct of P w.r.t. the model M_2, P^{M_2} is

$$P^{M_2} = \{b \leftarrow c.$$

$$c\}.$$

Again, $lm(P^{M_2}) = \{b, c\}$. Since, $lm(P^{M_2}) = M_2$, M_2 is an answer set of P. △

Example 8. *Consider a normal logic program P with rules*

$$a \leftarrow not(b).$$

$$b \leftarrow not(c).$$

$$c \leftarrow not(a).$$

The program P has no answer sets. To show this, we note that P has the following models: $M_1 = \{a, b, c\}$, $M_2 = \{a, b\}$, $M_3 = \{a, c\}$, and $M_4 = \{b, c\}$. We compute here the reducts of P w.r.t. each of these models:

$$P^{M_1} = \emptyset \neq M_1$$

$$P^{M_2} = \{b.\}$$

$$P^{M_3} = \{a.\}$$

$$P^{M_4} = \{c.\}$$

Clearly, in each case $lm(P^{M_i}) \neq M_i$. $\triangle$

Example 9. *Consider a Horn logic program P with rules*

$$\{a \leftarrow b, c.$$

$$b \leftarrow c.$$

$$b \leftarrow d.$$

$$c.\}$$

This program P has a unique answer set $M = \{a, b, c\}$ which is $lm(P^M)$. $\triangle$

Definition 2. *A set of atoms M is an answer set of a logic program with constraints P if M is an answer set of the program P without constraints, and M models every constraint in P.*

Theorem 10. *Every Horn logic program P has a unique* answer set *and it is the least model $lm(P)$.*

Proof. Let P be a Horn logic program. Then we know by the definition of a Horn logic program that P is negation free. Since P is negation free, the reduct of P w.r.t to any model M of P results in P (i.e., $P^M = P$). We also know that since P is Horn it has a unique least model M'. Hence, for any model M of P where $M \neq M'$, we clearly observe that M is not an answer set of P, since $lm(P^M) = lm(P) = M'$. However, M' is an answer set since it is a model of P and $lm(P^{M'}) = lm(P) = M'$. Thus P has a unique answer set which is its least model M'.

□

2.2.3 Supported models

Definition 3. *Let P be a logic program. For any set of atoms $M \subseteq At(P)$:*

- *An atom a is* supported *by P and M, if there exists a rule $r \in P$, where $head(r) = a$, $posbody(r) \subseteq M$ and $negbody(r) \cap M = \emptyset$ (i.e., M satisfies $body(r)$).*
- *M is* supported *under P, if for every atom $a \in M$, a is supported by P and M.*
- *M is a* supported model *of P, if M is a model of P and M is supported under P.*

Proposition 1. *[54] Let M be an answer set of a normal logic program P, then M is a supported model of P.*

We show here that the answer sets M_1 and M_2 of the program in Example 7 are supported models. The atom $a \in M_1$ is supported by the rule $a \leftarrow not(b)$, since $b \notin M_1$. The atom $c \in M_1$ is supported by the rule c. Hence, since both the atoms in model M_1 are supported by the program in Example 7, M_1 is a supported under P. Since M_1 is a model of P and M_1 is supported under P, M_1 is a supported model of P. Similarly, one can observe that M_2 is also a supported model. This is consistent with Proposition 1. We also note that the program $P = \{a \leftarrow a\}$ has two supported models: $\emptyset$ and $\{a\}$. Only the first of them is an answer set of P. Thus, the converse to Proposition 1 does not hold.

In the program in Example 8 the model M_1 contains all atoms in the program. Since none of the atoms in M_1 can be supported by any rule in the program w.r.t. M_1 and P, M_1 is not a supported model. The other models M_2, M_3 and M_4 of the program in Example 8

are also not supported models, since, we observe that in each model exactly one of the two atoms is not supported by any rule w.r.t. to the model and the program.

We next consider the notion of a *completion* of a logic program. The completion of a logic program results in a boolean formula in propositional logic whose satisfying boolean assignments (i.e., models) are in one-to-one correspondence with supported models.

The completion is based on the intuition that a logic program rule can be viewed as a definition of an atom appearing in the head of its rules. Hence, the completion enforces this definition by defining an equivalence relation between every atom a and the disjunction of the bodies of the rules with the atom a in its head.

2.2.4 Completion of a logic program

Let P be a logic program and let At be the set of atoms in P. For each atom $a \in At$, let $R_a = \{r \in P | head(r) = a\}$. Let us use $bd(r)$ to denote the conjunction of the literals $b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m$ in a rule of the form $a_i \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m)$.

Then, we define

$$C_a = \bigvee_{r \in R_a} bd(r).$$

If an atom a is not present in the head of any rule, then C_a is an empty disjunction and so, it is a contradiction denoted by $\bot$. The Clark's completion [28] of a logic program P denoted by P_{Comp} is obtained as follows,

$$P_{Comp} = \{a \leftrightarrow C_a : a \in At\}.$$

Example 11. *Let us consider a logic program P with rules*

$$\{a \leftarrow not(b).$$

$$b \leftarrow c,\ not(a), not(d).$$

$$a \leftarrow c.\}$$

Then its completion P_{Comp} is

$$\{a \leftrightarrow \neg b \vee c.$$

$$b \leftrightarrow c \wedge \neg a \wedge \neg d.$$

$$d \leftrightarrow \bot .$$

$$c \leftrightarrow \bot .\}$$

$\triangle$

Theorem 12. *[28] If P is a normal logic program and M is an answer set of P, then M is a model of P_{comp}.*

However, the converse of the above theorem does not hold as can be seen from the example show here.

Example 13. *Let P be a logic program:*

$$P = \{a \leftarrow not(b), c$$

$$c \leftarrow not(b), a\}.$$

Then

$$P_{comp} = \{a \leftrightarrow \neg b \wedge c.$$

$$c \leftrightarrow \neg b \wedge a.$$

$$b \leftrightarrow \bot .\}$$

We can observe that P_{comp} has a model $M = \{c, a\}$. The reduct

$$P^M = \{a \leftarrow c.$$

$$c \leftarrow a.\},$$

and $lm(P^M) = \emptyset \neq M$. Hence, M is not an answer set of P.

$\triangle$

Theorem 14. *[54] Let P be a logic program. Then, M is a supported model P iff M is a model of its completion P_{Comp}.*

We introduce here a class of logic programs for which the converse of the theorem holds.

2.2.5 Tight logic programs

Let P be a normal logic program with a set $At(P)$. Then P is *tight* if there is a mapping Φ

$$\Phi : At \rightarrow \{1, \ldots\}$$

such that for each rule $r \in P$ of the form

$$a \leftarrow b_1, \ldots, b_k, not(c_1), \ldots, not(c_m),$$

we have

$$\Phi(a) > \max\{\Phi(b_1), \ldots, \Phi(b_k)\}.$$

A program P is *non-tight* if such a mapping Φ does not exist.

Theorem 15. *[28] Let P be a tight logic program and $M \subseteq At(P)$, then M is an answer set of P iff M is a model of its completion P^C.*

However, if P is non-tight then the assertion may fail as can be seen in the example shown here.

Example 16. *Let P be a logic program:*

$$\{a \leftarrow b.$$

$$b \leftarrow a.\}$$

Then, its completion P_{comp} is

$$\{a \leftrightarrow b.$$

$$b \leftrightarrow a.\}$$

There are two models of the completion $M_1 = \emptyset$, and $M_2 = \{a, b\}$. Here, $lm(P^{M_1}) = \emptyset$, and $lm(P^{M_2}) = \emptyset$. Clearly, M_1 is an answer set of P, and M_2 is not.

△

Lin and Zhao restore the above theorem for non-tight programs by strengthening the concept of the completion. They introduce a specially structured class of propositional

formulas called *loop formulas* [49]. They show that the addition of loop formulas to the completion of logic programs results in a theory whose models correspond to answer sets of the logic program. We need to introduce first the concept of *positive dependency graph* before we explain the notion of loop formulas.

2.2.6 Positive dependency graph

A positive dependency graph of a logic program P denoted by $G(P)$ is a directed graph that is defined in the following way.

- The set $At(P)$ of atoms in the logic program P is the set of nodes in the graph.
- For every rule $r \in P$ of the form

$$a \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m),$$

  there is a directed edge from a to b_i, $i = 1, 2, \ldots, n$.

2.2.7 Loop formulas

We introduce here the concept of loop formulas [49] for logic programs.

Definition 4. *A set of atoms $L \subseteq At(P)$ is called a loop of a logic program P if the subgraph of the positive dependency graph $G(P)$ that is induced by L is strongly connected. A singleton atom is a loop if there is an edge from the atom to itself in $G(P)$.*

Example 17. *The dependency graph $G(P)$ of a program $P =$*

$$\{a \leftarrow b, not(c).$$

$$b \leftarrow a, not(c).\}$$

is shown in Figure 2.2. The subgraph induced by $\{a, b\}$ is strongly connected. Thus, $\{a, b\}$ is a loop. We note that $\{a, b\}$ is a model of the completion, which consists of formulas $\{a \leftrightarrow b \wedge \neg c, b \leftrightarrow a \wedge \neg c, c \leftrightarrow \bot\}$ (equivalently, $\{a, b\}$ is a supported model of P). $\triangle$

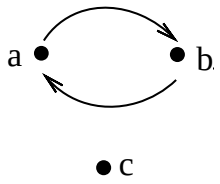


Figure 2.2: Positive dependency graph $G(P)$

We note that for the case of the program from Example 17, none of the atoms in the loop $\{a, b\}$ has a rule that could support it without depending positively on an atom in the loop. In other words, all possible ways of deriving atoms in the loop depend positively on other atoms in the loop. Lin and Zhao [49] observe that, if M is a model of the completion of P (equivalently, a supported model of P), and M is not an answer set of P, then M contains such a self-justifying loop. In Example 17, a model $\{a, b\}$ of the completion contains a loop $\{a, b\}$ that is self-justified. We will now provide a more complex illustration of this observation.

Example 18. *Let* $P_1 =$

$$\{a \leftarrow b, not(c).$$

$$b \leftarrow a, not(c).$$

$$b \leftarrow not(c).$$

$$c \leftarrow d.$$

$$d \leftarrow c.\}$$

The completion of the example, $P_{1_{comp}}$ *is*

$$\{a \leftrightarrow b \wedge \neg c.$$

$$b \leftrightarrow a \wedge \neg c \vee \neg c.$$

$$c \leftrightarrow d.$$

$$d \leftrightarrow c.\}.$$

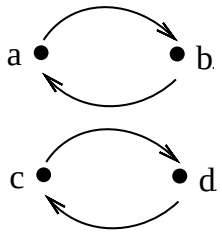


Figure 2.3: Positive dependency graph $G(P_1)$

The dependency graph $G(P_1)$ is shown in Figure 2.3. We note that there are two loops: $\{a, b\}$ and $\{c, d\}$. The latter is self-justifying. But the former is not. There is a rule that potentially could support b (and so also other elements in the loop) that does not depend positively on other atoms in the loop. Namely, the rule $b \leftarrow not(c)$ has these properties.

We note that both $\{a, b\}$ and $\{c, d\}$ are supported models of P_1 but only $\{a, b\}$ is an answer set.

△

The authors model these loops as formulas so that models of the union of the program completion and the formulas of all loops are in one-to-one correspondence with the answer sets of the logic program as shown here. Given a loop L of a program P, Lin and Zhao defined a loop formula F_L so as to capture the idea of "external" support. Namely, let L be a loop of a program P, and a subset of rules R_L of P be given by

$$R_L = \left\{ \; r = a \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m) \quad | \quad r \in P, a \in L, \bigvee_{i=1}^{n} b_i \notin L \; \right\}.$$

Let us use $bd(r)$ to denote the conjunction of the atoms $b_1, \ldots, b_n$, and the negated atoms $\neg c_1, \ldots \neg c_m$ that appear in $body(r)$.

.

Definition 5. *Let L be a loop and let $R_L = \{r_L^1, \ldots, r_L^n\}$. A loop formula F_L is an implication:*

$$\begin{cases} \bigvee_{a \in L} a \Rightarrow \bigvee_{r \in R_L} bd(r) & \text{if } R_L \neq \emptyset \\ \bigvee_{a \in L} a \Rightarrow \bot & \text{if } R_L = \emptyset \end{cases}$$

Example 19. *In Example 18 we have two loops* $L_1 = \{a, b\}$ *and* $L_2 = \{c, d\}$*, and the corresponding set of rules*

$$R_{L_1} = \{b \leftarrow not(c)\},$$

and

$$R_{L_2} = \emptyset.$$

The corresponding loop formula for L_1 *is,*

$$F_{L_1} = a \vee b \Rightarrow \neg c$$

$$\equiv (\neg a \wedge \neg b) \vee \neg c$$

$$\equiv (\neg a \vee \neg c) \wedge (\neg b \vee \neg c),$$

and for L_2 *is*

$$F_{L_2} = c \vee d \Rightarrow \perp$$

$$\equiv \neg c \wedge \neg d.$$

We can observe that $P_{comp} \cup F_{L_1} \cup F_{L_2}$ *has only one model* $M_1 = \{a, b\}$ *which is the only answer set of the program from Example 18.*

△

Lin and Zhao showed that if the completion of the program P is expanded with all loop formulas for P, then models of the resulting theory are precisely answer sets of the program — those supported models that are self-supported are eliminated. Lin and Zhao proved the following theorem.

Theorem 20. *[49] Let* P *be a logic program,* M *is an answer set of a logic program iff* M *is a model of the union of the completion of the program* P *and loop formulas of all loops in* P*.*

The theorem is important as it provides us a way to directly translate between logic programs and propositional SAT theories, thus allowing ASP solvers to exploit in their design the techniques used in the implementation of SAT solvers. It also provides a theoretical foundation for several search prunings used by current ASP solvers.

2.2.8 Complexity

First, the decision problem of checking if a set of atoms M is an answer set of a normal logic program is in P and the problem of deciding if a normal logic program P has an answer set is NP-complete ([55]).

Next we consider special subclasses of logic programs. The problem of deciding if a class of normal logic programs which has rules with a fixed number of literals K where $K \geq 2$ is NP-complete. The problem of checking if a logic program of simple two literal rules wherein each rule consists of a single atom in the head and a single negated atom in its body of the form $a \leftarrow not(b)$ is NP-complete [55].

Finally, the problem of deciding if a Horn logic program has an answer set is in P follows from the result we have earlier in Page 22.

2.2.9 Solvers for logic programs

A program that computes answer sets of a logic program is called a solver or an ASP solver. There are many state-of-the-art solvers for computing answer sets of a logic program. Some of the solvers that are being used are *clasp* [32], *smodels* [62], *cmodels* [35, 47], *DLV* [25], *ASSAT* [49], and *pbmodels* [50]. These solvers can be categorized into two classes. The first class consists of solvers that are native to the area of answer set programming. *Smodels*, *DLV*, and *clasp* are examples of solvers in the first class. The second class consists of solvers that translate logic programs into a CNF formula or pseudo-boolean (PB) theory[1] and use existing solvers for these formalisms. The solvers *ASSAT*, *cmodels*, and *pbmodels* belong to the second class of solvers.

Smodels like all other native solvers computes answer sets of a logic program by performing a backtracking search algorithm. *Smodels* uses the computation of a well-founded model as a heuristic to guide the search. *Smodels* takes as input a ground normal logic program. The ground logic program provided to *smodels* can be obtained by using a grounder called *Lparse* [74]. *Lparse* produces the ground logic program in the format accepted by *smodels*.

[1] It is a system of linear inequalities with boolean variables and integer coefficients.

The *DLV* system is another native ASP system that computes the answer sets of a disjunctive logic program (i.e., logic programs whose rules have an extended syntax). The *DLV* system computes the answer sets of the logic program in two steps. The *DLV* system uses a program called the model generator to guess a candidate for the answer set of the logic program, and then it uses a program called a model checker to check if it is an answer set.

Clasp is a new ASP solver that computes answer sets of extended normal logic programs. *Clasp* incorporates recent advances in boolean constraint propagation techniques from the area of CSP and SAT. Some of these techniques include conflict-driven clause learning [73], nogood recording and deletion [73], restarts [37], and watched literals [58]. *Clasp* accepts logic programs that are in the format obtained after grounding a logic program with variables using grounders *lparse* and *GrinGo* [33].

ASSAT computes an answer set of a logic program by using a SAT solver as a back-end engine. The solver first computes the completion of the program and feeds it to a SAT solver which generates a model, if one exists. Each model generated by the SAT solver is checked to see if it is an answer set. If so *ASSAT* stops, else *ASSAT* appends an appropriate loop formula to the completion theory and re-runs the SAT formula on the new theory. The loop formulas prevent the re-generation of models of the completion theory that are not answer sets. However, ASSAT has a possibility of adding an exponential number of such loop formulas. *ASSAT* takes as input a ground normal logic program in the format produced by *lparse*, and a complete SAT solver that is to be used as the back-end engine for model generation.

The solver *cmodels* also computes the answer set of a logic program using a procedure similar to the one described above for *ASSAT*. However, in *cmodels* the generation of an answer set is integrated within the SAT solver. The completion of the program is provided to the SAT solver, which generates models of the completion. Each model is tested to see if it is an answer set of the program. If so, then *cmodels* outputs it, otherwise it computes loop formulas. *Cmodels* uses this formula to backtrack to a node in the search tree of the SAT solver and continues to generate other models. *Cmodels* also differs from *ASSAT* in

that it can compute answer sets of logic programs with extended rules.

The solver *pbmodels* translates logic programs extended with weight rules to a PB-theory and uses PB solvers[2] to compute answer sets [50]. This solver was based on the completion of the logic program extended with weight rules into a propositional logic theory extended with weight atoms. The authors also extended the concept of loop formulas to propositional theories with weight atoms so that answer sets of the logic program correspond to models of the propositional theory with weight atoms [51]. *Pbmodels* takes as input a normal logic program extended with choice, cardinality, and weight rules in the format produced by *lparse*, performs its completion into the language of propositional logic extended with weight atoms, and finally translates the propositional logic theory into PB-theories that can be accepted by various PB solvers.

[2]They are programs designed to find solutions to a PB-theory.

Chapter 3

Random Logic Programs

In this chapter we provide models of random logic programs and study their properties. We consider random programs with rules of the same length, and in particular 2-literal rules. These 2-literal programs despite their simplicity, are of considerable interest due to the following. First, as we have noted earlier in Chapter 2, these 2-literal programs are NP-complete. Second, many problems of interest have a simple encoding in terms of such programs [41].

We study in this chapter experimental and analytical properties of random logic programs with two-literal rules. We begin by defining five different classes of programs each of which can be distinguished based on the types of 2-literal rules. We also consider programs from classes that are formed by a union of some or all of these five classes. Next, we provide a method for randomly generating a program from any of these classes. Finally, we analyze the properties of these different classes of randomly generated programs.

The first property we study is the probability of existence of an answer set for randomly generated logic programs from these classes. Then, we experimentally study the hardness of random programs for ASP solvers, by determining the average time taken and the average number of choice points generated by solvers to compute the existence of answer sets. We show that for logic programs that are *constraint-free* and *purely negative* an easy-hard-easy pattern emerges as we plot the average number of choice points generated by ASP solvers for randomly generated programs with increasing density of rules. We give arguments to explain that pattern, and show that the hardness of programs from the hard region grows quickly with the number of atoms. Our results point to the importance of constraint-free purely negative programs for the development of ASP solvers, as they can serve as useful benchmarks when developing good search heuristics.

The organization of the sections of this chapter is as follows. Section 3.1 provides an introduction to the different classes of 2-literal programs. The experimental and theoretical

analysis of the probability of existence of an answer set for random programs from these classes is provided in Section 3.2. Finally, Section 3.3 gives the experimental results on the difficulty of these programs for ASP solvers, as well as the theoretical results supporting our explanation for the existence of an easy-hard-easy phenomenon.

Our work presented in this chapter appears in Proceedings of LPNMR-09 [59].

3.1 2-Regular Programs

We assume a fixed set of atoms $At = \{a_1, a_2, \ldots\}$. There are five types of 2-regular rules:

$$a \leftarrow not(b);$$

$$a \leftarrow b;$$

$$\leftarrow not(a), not(b);$$

$$\leftarrow a, not(b);$$

$$\leftarrow a, b.$$

Accordingly, we define five classes of programs, mR_n^-, mR_n^+, mC_n^-, $mC_n^\pm$, and mC_n^+, with atoms from $At_n = \{a_1, \ldots, a_n\}$ and consisting of m rules of each of these types, respectively. Without the reference to m, the notation refers to all programs with n atoms of the corresponding type (for instance, R_n^+ stands for the class of all programs over At_n consisting of proper rules of the form $a \leftarrow b$).

The maximum value of m for which mR_n^-, mR_n^+ and $mC_n^\pm$ are not empty is $n(n-1)$. The maximum value of m for which mC_n^- and mC_n^+ are not empty is $n(n-1)/2$. Let $0 \leq m_1, m_2, c_2 \leq n(n-1)$ and $0 \leq c_1, c_3 \leq n(n-1)/2$ be integers. By $[m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$ we denote the class of programs P that are unions of programs from the corresponding classes. We refer to these programs as *components* of P. If any of the integers m_i and c_i is 0, we omit the corresponding term from the notation. We provide here a few examples.

Example 21. *The following program P belongs to the class* $[2R^- + 2R^+ + 2C^- + 1C^{\pm} + 1C^+]_{10}$.

$$P = \{a_6 \leftarrow not(a_2).$$

$$a_3 \leftarrow not(a_5).$$

$$a_4 \leftarrow a_1.$$

$$a_3 \leftarrow a_7.$$

$$\leftarrow not(a_1), not(a_9).$$

$$\leftarrow not(a_3), not(a_8).$$

$$\leftarrow a_2, not(a_{10}).$$

$$\leftarrow a_9, a_5.\}$$

△

Example 22. *Here we present a program P from the class* $[1R^- + 2C^- + 1C^{\pm} + 1C^+]_{10}$

$$P = \{a_1 \leftarrow not(a_3).$$

$$\leftarrow not(a_1), not(a_9).$$

$$\leftarrow not(a_3), not(a_8).$$

$$\leftarrow a_2, not(a_{10}).$$

$$\leftarrow a_9, a_5.\}$$

△

When we do not specify the numbers of rules, we allow any programs from the corresponding classes. For instance, $[R^- + R^+ + C^- + C^{\pm} + C^+]_n$ stands for the class of all proper programs with atoms from At_n.

Given integers n and m, it is easy to generate uniformly at random programs from each class mR_n^-, mR_n^+, mC_n^-, $mC_n^{\pm}$, and mC_n^+. For instance, a random program from mR_n^- can be viewed as the result of a process in which we start with the empty program on

the set of atoms At_n, and then in each step we add a randomly generated proper rule of the form $a \leftarrow not(b)$, with repeating rules discarded, until m rules are generated. In the case when m is close to its maximum value for its class, denoted by m_{max}, we generate a random program by starting with a program that has all possible rules in its class. We then randomly choose a rule from the program to discard, and remove the rule from the program. We repeatedly choose $m_{max} - m$ rules from the program and discard them. This approach generalizes easily to programs from other classes we consider, in particular, to programs from $[m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$. Our goal is to study properties of such random programs.

Proposition 2. *Let $P \in [m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$ ($m_1 = 0$). If $c_1 = 0$ then P has a unique stable $\emptyset$. If $c_1 > 0$ then P has no answer set.*

Proof. We note here that the program P is a union of a Horn program $H \in [m_2R^+]_n$ and set of constraints $C \in [c_1C^- + c_2C^\pm + c_3C^+]_n$. Since H is a Horn program, it has a unique answer set M, which happens to coincide with the least model of H. Since every rule in H has the form $a \leftarrow b$, it is clear that $M = \emptyset$ is the least model of H and so, a unique answer set of H.

Now, let us assume that $c_1 = 0$, and consider any constraint $c \in C$. Then, $c = \leftarrow a,\ not(b)$ or $c = \leftarrow a,\ b$. Since, c has at least one non-negated occurrence of an atom in the body of c and $M = \emptyset$, $M \models c$. Thus $M \models P$ and M is a unique answer set of P.

Finally, let us assume that $c_1 > 0$. Then C contains a constraint of the form $\leftarrow not(a),\ not(b)$. Since $\{a, b\} \notin M$, $M \not\models c$. Consequently, M is not an answer set of P and so, P has no answer sets. □

Thus, in order to obtain interesting classes of programs, we must have $m_1 > 0$. In other words, programs from R_n^- (proper purely negative and constraint-free) play a key role.

3.2 The Probability of a Program to Have an Answer Set

We study first the probability that a random program in the class $[m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$ has an answer set. In several places we use results from random graph

theory [13, 42]. To this end, we exploit graphs associated with programs. Namely, with a program $P \in [R^- + R^+ + C^\pm]_n$ we associate a *directed* graph $D(P)$ with the vertex set At_n, in which a is connected to b with a directed edge (a, b) if $b \leftarrow not(a)$, $b \leftarrow a$ or $\leftarrow b, not(a)$ is a rule of P. For $P \in [R^- + R^+]_n$, the graph $D(P)$ is known as the *dependency* graph of a program. Similarly, with a program $P \in [R^- + R^+ + C^- + C^\pm + C^+]_n$ we associate an undirected graph $G(P)$ with the vertex set At_n, in which a is connected to b with an *undirected* edge $\{a, b\}$ if a and b appear together in a rule of P. If $P \in [R^- + R^+ + C^\pm]_n$, then $D(P)$ may have fewer edges than P has rules (the rules $a \leftarrow not(b)$, $a \leftarrow b$ and $\leftarrow b, not(a)$ determine the same edge). A similar observation holds for $G(P)$.

These graphs contain much information about the underlying programs. For instance, it is well known that if $P \in [R^- + R^+]_n$ and $D(P)$ has no cycles then P has a unique answer set. We illustrate this property here.

Example 23. *Let* $P =$

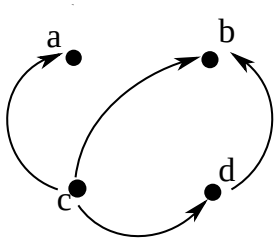
$$\{d \leftarrow not(c).$$

$$a \leftarrow c.$$

$$b \leftarrow d.$$

$$b \leftarrow not(c).\}.$$

Then, the dependency graph $D(P)$ *shown here has no cycles, and* $M = \{b\}$ *is the unique answer set of* P.



△

Next, let us assume an answer set M of a program P from the class $[m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$. Then we can show that elements in its complement set $\overline{M}$ form an independent set in the graph $G(P_1)$. We show this property in the proposition given below.

Proposition 3. *Let $P \in [m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$ and M be an answer set of P. Then $\overline{M}$ is an independent set in the graph $G(P_1)$, where P_1 is the component of P from $m_1R^-_n$.*

Proof. Let $At(P)$ be the set of atoms in P. We know that M is an answer set of P. Let us assume that $\overline{M} = At(P) \setminus M$ is not an independent set in the graph $G(P_1)$. Then, there must exist a pair of vertices a and b such that $\{a, b\} \subseteq \overline{M}$ and a and b are connected by an undirected edge (a, b) in $G(P_1)$. Since (a, b) is an edge in $G(P_1)$ it implies that at least one of the rules $a \leftarrow not(b)$ or $b \leftarrow not(a)$ is in P_1. First, let us assume that $r = a \leftarrow not(b)$ is in P_1. Then since M is an answer set of P, M is a model of $a \leftarrow not(b)$. But, $b \notin M$, and $a \notin M$, which implies that $M \not\models r$, a contradiction. Secondly, let us assume that $r = b \leftarrow not(a)$ is in P_1. Then since M is an answer set of P, M must be a model of $b \leftarrow not(a)$. But, $a \notin M$, and $b \notin M$, which implies that $M \not\models r$, is a contradiction. Hence, $\overline{M}$ is an independent set in the graph $G(P_1)$. □

We observe that if $P \in [R^- + R^+ + C^\pm]_n$, then $D(P)$ may have fewer edges than P has rules (the rules $a \leftarrow not(b)$, $a \leftarrow b$ and $\leftarrow b, not(a)$ determine the same edge). The same holds for the graph $G(P)$. However, if P is a program drawn uniformly at random from $[m_1R^- + m_2R^+ + c_2C^\pm]_n$, then $D(P)$ can be regarded as a *subgraph* of a random directed graph with n vertices and $m = m_1 + m_2 + c_2$ edges (loops disallowed). Indeed, we can view $D(P)$ as the result of a process in which we insert randomly selected edges into the graph we construct, with repeating edges discarded (it is because, P can be constructed by a similar process). Upon constructing $D(P)$, if $D(P)$ has fewer edges than m, we simply continue the process until m edges are generated. The result is a random graph D' with n atoms and m edges that is a *supergraph* for $D(P)$. Similarly, if $P \in [m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$, then $G(P)$ can be viewed as a subgraph of a random graph with n vertices and m edges.

We denote by $\mathcal{AS}^+$ the class of all programs over At that have answer sets. We write $Prob(P \in \mathcal{AS}^+)$ for the probability that a random graph P from one of the classes defined above has an answer set. That probability depends on n (technically, it also depends on the numbers of rules of particular types, but whenever it is so, the relevant numbers are themselves expressed as functions of n). We are interested in understanding the behavior of $Prob(P \in \mathcal{AS}^+)$ for random programs P from the class $[R^- + R^+ + C^- + C^{\pm} + C^+]_n$ (or one of its subclasses). More specifically, we will investigate $Prob(P \in \mathcal{AS}^+)$ as n grows to infinity. If $Prob(P \in \mathcal{AS}^+) \to 1$ as $n \to \infty$, we say that P *asymptotically almost surely*, or *a.a.s* for short, has answer sets. If $Prob(P \in \mathcal{AS}^+) \to 0$ as $n \to \infty$, we say that P a.a.s. has no answer sets.

To provide intuitions for our results, we first consider the probability that a program from mR^-_{150} has an answer set as a function of the density $d = m/150$ (or equivalently, the number of edges m). The graphs, shown in Figure 3.2 and 3.1, were obtained experimentally. For each value of d, we generated 1000 graphs from the set mR^-_{150}, where $m = 150d$. The graph in Figure3.2 shows the behavior of the probability across the entire range of d. The graph in Figure 3.1 shows in more detail the behavior for small densities.

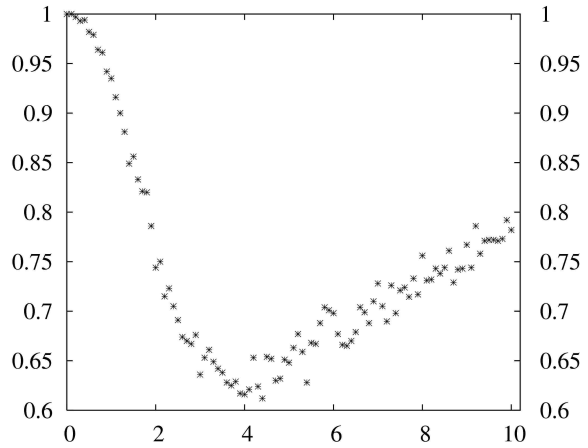


Figure 3.1: The probability that a graph from mR^-_{150} ($m = 150d$) has an answer set, as a function of d.

We start with programs of low density and assume first that they do not have constraints. In this case, the results do not depend on whether or not we allow positive rules.

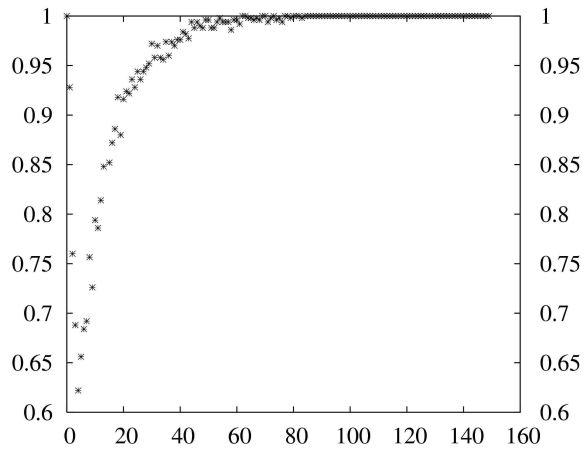


Figure 3.2: The probability that a graph from mR^-_{150} ($m = 150d$) has an answer set, as a function of d.

Theorem 24. *If $m_1 + m_2 = o(n)$ and $P \in [m_1R^- + m_2R^+]_n$, then P a.a.s has a unique answer set.*

Proof. Let P be a random program from $[m_1R^- + m_2R^+]_n$, and $m = m_1 + m_2$. The directed graph $D(P)$ can be viewed as a subgraph of a random directed graph with n vertices, and $m' = o(n)$ edges ($m' \leq m$, as different rules in P may map onto the same edge). Thus, $D(P)$ a.a.s. has no directed cycles (the claim follows from the property of random *undirected* graphs: a random undirected graph with n vertices and $o(n)$ edges a.a.s. has no cycles [42]). Thus P a.a.s. has a unique answer set. □

If there are constraints in the program, the situation changes. Even a single constraint of the form $\leftarrow not(a), not(b)$ renders a sparse random program inconsistent.

Corollary 25. *If $c_1 \geq 1$, $m_1 + m_2 = o(n)$, and P is a random program from $[m_1R^- + m_2R^+ + c_1C^-]_n$, then P a.a.s. has no answer sets.*

Proof. Let P be a random program from $[m_1R^- + m_2R^+ + c_1C^-]_n$. Then, $P = P_1 \cup P_2$, where P_1 is a random program from $[m_1R^- + m_2R^+]_n$ and P_2 is a random program from $c_1C^-_n$. By Theorem 24, P_1 a.a.s. has a unique answer set, say M. Since P_1 has $o(n)$ non-constraint rules, $|M| = o(n)$. The probability that a randomly selected constraint of the form $\leftarrow not(a), not(b)$ is violated by M is given by the probability that $\{a, b\} \cap M = \emptyset$.

Since, there are $o(n)$ possible atoms that can appear in the head of any rule as well as in M, we have $n-o(n)$ atoms that do not appear in the head of a rule. These $n-o(n)$ atoms cannot appear in any answer set M as they will not be supported by any rule in P. Thus, there are at least $\binom{n-o(n)}{2}$ constraints of the form $\leftarrow not(a), not(b)$ such that $\{a,b\} \cap M = \emptyset$. Thus the probability that a randomly selected constraint of this type is violated by M is at least

$$\frac{\binom{n-o(n)}{2}}{\binom{n}{2}}$$

which is expanded as,

$$\frac{(n-o(n))(n-o(n)-1)}{n(n-1)}.$$

On dividing the top and bottom of the fraction by n^2, the probability is

$$\frac{(1-\frac{o(n)}{n})(1-\frac{o(n)}{n}-\frac{1}{n})}{(1-\frac{1}{n})}.$$

Since for any function $f(n) \in o(n)$, we know from the definition of $o(n)$ that $f(n)/n \to 0$ as $n \to \infty$, and it follows that the probability computed above converges to 1 as $n \to \infty$. Thus, the assertion follows. □

If we exclude such constraints then there is again a small initial interval of densities, for which random programs are consistent with high probability.

Corollary 26. *If $c_1 = 0$, $c_2 + c_3 \geq 1$, $(m_1+m_2)c_2 = o(n)$, $(m_1+m_2)^2 c_3 = o(n^2)$, and P is a random program from $[m_1R^- + m_2R^+ + c_2C^{\pm} + c_3C^+]_n$, then P a.a.s. has an answer set.*

Proof. Let P be a random program from $[m_1R^- + m_2R^+ + c_2C^{\pm} + c_3C^+]_n$. Thus, $P = P_1 \cup P_2 \cup P_3$, where P_1, P_2 and P_3 are random programs from $[m_1R^- + m_2R^+]_n$, $c_2C_n^{\pm}$ and $c_3C_n^+$, respectively. Since $c_2 > 0$ or $c_3 > 0$, $m_1 + m_2 = o(n)$. By Theorem 24, P_1 a.a.s. has a unique answer set, say M. Moreover, the size of M is at most $m_1 + m_2$. First, the probability that a randomly selected constraint of the form $\leftarrow a, b$ is violated by M, is given by the probability that both $a \in M$ and $b \in M$. This probability is

$$\frac{\binom{|M|}{2}}{\binom{n}{2}}.$$

The probability that at least one such rule in P is violated by M is

$$\leq c_3 \frac{|M|(|M|-1)}{n(n-1)},$$

which is

$$\leq c_3 \frac{(m_1+m_2)^2}{n(n-1)}.$$

Since $\frac{c_3(m_1+m_2)^2}{n^2} \to 0$ as $n \to \infty$, this probability converges to 0. Second, the probability that a randomly selected constraint of the form $\leftarrow a, not(b)$ is violated by M, is given by the probability that both $a \in M$ and $b \notin M$. This probability is

$$\frac{|M|(n-|M|)}{n(n-1)}.$$

The probability that at least one such rule in P is violated by M is

$$\leq c_2 \frac{|M|(n-|M|)}{\binom{n}{2}},$$

which is

$$\leq c_2 \frac{(m_1+m_2)(n-(m_1+m_2))}{n(n-1)}.$$

Since $\frac{c_2(m_1+m_2)}{n} \to 0$ as $n \to \infty$, this probability converges to 0.

Thus, a.a.s. programs P_2 and P_3 are satisfied by M. Consequently, P a.a.s. has M as its unique answer set of P. □

We move on to programs of high density. The first result concerns programs from R_n^- (proper, purely negative, and constraint-free programs with n atoms).

Theorem 27. *[Truszczyński,[59]] Let $0 < c < 1$ be a constant. For every fixed x, a random program from mR_n^-, where $m = \lfloor cN + x\sqrt{c(c-1)N} \rfloor$ and $N = n(n-1)$, a.a.s. has an answer set.*

Theorem 27 concerns only a narrow class of dense programs, its applicability being limited by the specific number of rules programs are to have ($m = \lfloor cN + x\sqrt{c(c-1)N} \rfloor$,

where $N = n(n-1)$). It also does not apply to "very" dense graphs with $m = n^2 - o(n^2)$ rules. However, based on that theorem and on experimental results (Figure 3.2), we conjecture that for every $c > 0$, a program from mR_n^-, where $m \geq cn^2$, a.a.s. has an answer set.

We will now consider the effect of adding positive rules (rules of the form $a \leftarrow b$) and constraints. In fact, as soon as we have just slightly more than $n \log n$ positive rules in a random program that program a.a.s. has no answer sets.

Theorem 28. *For every $\epsilon > 0$, if $m_1 \geq 1$, $m_2 \geq (1+\epsilon)n \log n$, and P is a random program from $[m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$, then P a.a.s. has no answer sets.*

Proof. Let $P \in [m_1R^- + m_2R^+ + c_1C^- + c_2C^\pm + c_3C^+]_n$, where $m_1 \geq 1$. Also, let P_2 be the component of P from m_2R^+. If $D(P_2)$ contains a Hamiltonian cycle, then P has no answer sets. Indeed, $\emptyset$ is not an answer set due to the rule of the form $a \leftarrow not(b)$ that is present in P. Thus, if P has an answer set, say M, then $M \neq \emptyset$. Clearly, P^M contains P_2. By the assumption on $D(P_2)$, every vertex can be reached from any vertex through the edges in the hamiltonian. Similarly, corresponding to each directed edge (a, b) in the hamiltonian cycle there is a rule of the form $r = b \leftarrow a$. If $a \in M$ in any model of P where $r \in P$, then it implies that $b \in M$. Hence, since every vertex can be reached by an edge starting from a single vertex in the hamiltonian cycle, every atom in the program must be true in M if even a single atom is true in M. Hence, the least model of P^M contains all atoms in At_n. Thus, $M = At_n$. But then, P^M contains no atoms (all its rules are either from P_2 or are constraints of the form $\leftarrow a, b$) and so, the least model of P^M is $\emptyset$, a contradiction. Clearly, there is a precise correspondence between programs from m_2R^+ and random directed graphs with n nodes and m edges (no loops). The assertion follows now from the result that states that a random directed graph with n nodes and at least $(1+\epsilon)n \log n$ edges a.a.s. has a Hamiltonian cycle [13]. □

The presence of sufficiently many constraints of the form $\leftarrow a, b$ or $\leftarrow a, not(b)$ also eliminates answer sets. To see that, we first get the following result that provides a lower bound on the size of an answer set in a dense random logic program.

Theorem 29. *For every real $c > 0$, there is a real $d > 0$ such that a.a.s. the complement of every answer set of a random program $P = P_1 \cup P_2$, where $P_1 \in mR_n^-$, $P_2 \in [R^+ + C^- + C^\pm + C^+]_n$ and $m \geq cn^2$, has size at most $d \log n$.*

Proof. We recall that if M is an answer set of a program $P = P_1 \cup P_2$, where $P_1 \in R_n^-$ and $P_2 \in [R^+ + C^- + C^\pm + C^+]_n$, then M is the complement of an independent set in $G(P_1)$. The following property is useful here. For every real $c > 0$ there is a real $d > 0$ such that a.a.s. a graph with n vertices and $m \geq cn^2$ edges has no independent set with more than $d \log n$ elements [13]. Thus, the assertion follows. □

We now consider the effect of constraints of the form $\leftarrow a, b$ on the existence of answer sets in programs with many purely negative rules. Intuitively, even a small number of such constraints should suffice to "kill" all answer sets. Indeed, according to Theorem 29, these answer sets are large and contain "almost all" atoms.

Theorem 30. *[Truszczyński,[59]] For every $c > 0$ there is $d > 0$ such that if $m_1 \geq cn^2$, $c_3 \geq d \log n + 1$, and P is a random program from $[m_1 R^- + m_2 R^+ + c_1 C^- + c_2 C^\pm + c_3 C^+]_n$, then P a.a.s. has no answer sets.*

We see here the effect of constraints of the form $\leftarrow a, not(b)$ do not have such a dramatic effect. However, a still relatively small number of such constraints a.a.s. eliminates all answer sets.

Theorem 31. *[Truszczyński, [59]]For every $c > 0$, and for every $\epsilon > 0, if\ m_1 \geq cn^2$, $c_2 \geq n^{1+\epsilon}$, and P is a random program form $[m_1 R^- + m_2 R^+ + c_1 C^- + c_2 C^\pm + c_3 C^+]_n$, then a.a.s. P has no answer sets.*

The case of constraints $\leftarrow not(a), not(b)$ is less interesting. Large answer sets (having at least $n - d \log n$ atoms) that arise for programs with dense components from R_n^- typically satisfy them and to "kill" all answer sets of such programs with high probability almost all constraints $\leftarrow not(a), not(b)$ must be present.

3.3 Hardness of Programs

We will now study the hardness of programs from $[m_1R^- + m_2R^+ + c_1C^- + c_2C^{\pm} + c_3C^+]_n$ for ASP solvers. The bulk of our experimental results concern programs in the class R_n^-. It turns out these programs (for appropriately chosen density) are especially challenging.

Unless stated otherwise, our experiments separate programs that have answer sets (are *consistent*) from those that do not (are *inconsistent*). For each experiment we generate a sample of instances of programs of each of these two types. In the previous section we provided evidence that programs in mR_n^-, where $m \geq cn^2$ (cf. Figure 3.2 and Theorem 27), a.a.s. have an answer set. Therefore, when experimenting with inconsistent programs, we restrict the number of rules in a program to values for which inconsistent programs appear with probability sufficiently larger than 0 (about 0.05) to allow for building samples of inconsistent programs of sizes large enough to justify drawing conclusions from experiments (typically 100 programs per sample).

In experiments, we used *smodels* (with lookahead) and *clasp*. We took the average number of choice points as reported by these systems as the measure of the *hardness* of a family of programs.

Our first observation is that as we increase m, programs from mR_n^- show the easy-hard-easy pattern. That is, low-density programs are easy for the two solvers. When m grows, programs get harder. Then, at some point, they start getting easier again. We illustrate that behavior in Figure 3.3 and Figure 3.4 below. The two graphs show separately the results for consistent and inconsistent programs from the classes mR_{100}^-. Each figure shows together the results (average number of choice points) for *smodels* (the scale on the right) and *clasp* (the scale on the left). The x-axis shows the density, that is, the ratio of the number of rules to the number of atoms in a program. We stress that the scales differ. Thus, the figures are not meant to compare the performance of *smodels* and *clasp*. But they do show that for each solver a similar easy-hard-easy pattern emerges, and that the features of the pattern are remarkably similar for the two solvers.

We obtained the same type of pattern in our experiments with programs with 125, 175,

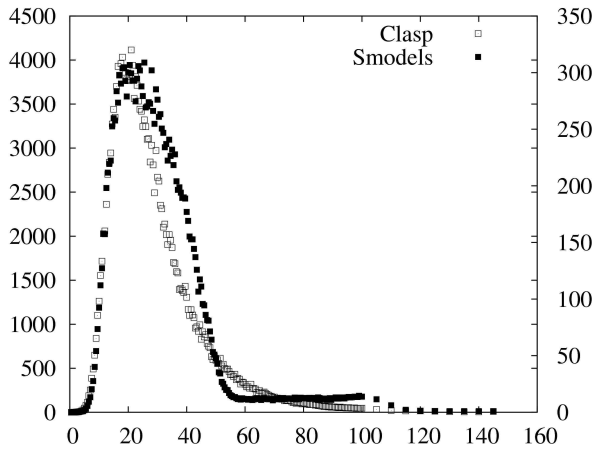


Figure 3.3: Average number of choice points for consistent programs with 150 atoms *smodels* (scale on the right) and *clasp* (scale on the left). The x-axis represents the density. Sample sizes are 500 for consistent programs, and 100 for inconsistent programs.

and 200 atoms. However, we observed some minor deviations from that pattern for *smodels* (but not for *clasp*) for programs with 100 atoms. Given our results for $n \geq 125$, it seems plausible that the irregular behavior arises only for some smaller numbers of atoms. We observe a similar easy-hard-easy pattern as we plot the time taken by both the solvers for programs from $[mR^{-}]_n$ with $n = 200$ and with increasing density $d = m/n$. We provide these results in Appendix A.

We used the term *hard region* above somewhat informally. To make that concept more precise, we define it here.

Definition 6. *The hard region is the maximum interval* $[u, v]$ *such that for every density* $d \in [u, v]$ *the average number of choice points is at least 90% of the maximum (peak) average number of choice points.*

Table 3.1 shows the hard regions, the density for which the number of choice points reaches the maximum, and the number of choice points at the peak location for consistent and inconsistent instances with $n = 125, 150, 175$, and 200 atoms. The key observations are:

1. the location of the hard region does not seem to depend much on the solver; it is centered around the density of 19 for consistent programs, and 22 for inconsistent

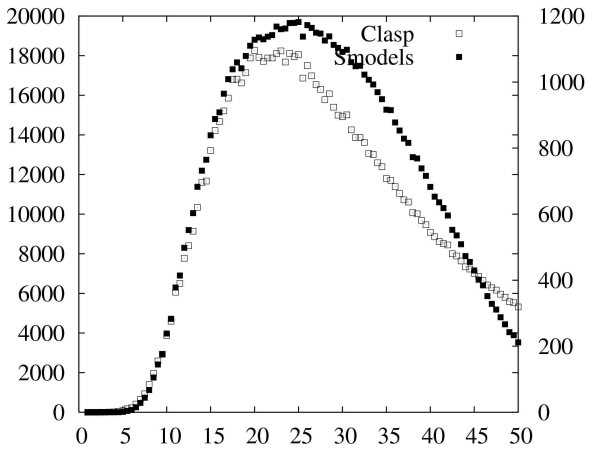


Figure 3.4: Average number of choice points for inconsistent programs with 150 atoms *smodels* (scale on the right) and *clasp* (scale on the left). The x-axis represents the density. Sample sizes are 500 for consistent programs, and 100 for inconsistent programs.

Table 3.1: Hard region, peak location, and the number of choice points at the peak location for consistent and inconsistent programs. Results for *clasp* and *smodels*.

Inconsistent programs						
	clasp			*smodels*		
n	hard region	peak	choice points at peak	hard region	peak	choice points at peak
125	$[17.5 - 27]$	22	5261	$[17.5 - 24]$	21	388
150	$[18 - 27]$	23	18639	$[19 - 31]$	24.5	1184
175	$[18.5 - 27.5]$	22	59704	$[17.5 - 23.5]$	20.5	3582
200	$[18 - 28]$	22	189576	$[18 - 26]$	22.5	14407
Consistent programs						
125	$[15.5 - 21.5]$	17.5	1231	$[16 - 25]$	20	130
150	$[16 - 23]$	17.5	4033	$[16 - 29.5]$	20	308
175	$[18.5 - 21.5]$	20	14230	$[17.5 - 21.5]$	20	1110
200	$[17.5 - 23]$	19.5	43345	$[18.5 - 24.5]$	19.5	4232

ones,

2. inconsistent programs are significantly harder than consistent ones,

3. the peak of hardness is not sharp, or, in other words, the hard region extends over a sizable range of densities, and

4. the hardness of programs in the hard region grows very quickly.

We conclude with arguments to explain the presence of the easy-hard-easy pattern we observed for programs in the class R_n^-. First, we note that programs in mR^-, where $m = o(n)$, a.a.s. are stratified (Theorem 24). Computing answer sets for such programs is easy.

As the density (the number of rules) grows, cycles in the graph $D(P)$ start appearing (that happens roughly when a program has as many rules as atoms). Initially, there are few cycles and the increase in hardness is slow. At some point, however, there are enough cycles in $D(P)$ to make computing answer sets of P hard. To explain why the task gets easier again, we note the following property of binary trees.

Proposition 4. *Let T be a binary tree with m leaves, the height n, and with the number of left edges on any path from the root to a leaf bounded by k. Then $m \leq 2^k \binom{n}{k}$.*

Proof. Let $S(n,k)$ be the maximum number of leaves in such a tree. Then $S(n,k)$ is given by the recursive formula $S(n,k) = S(n-1,k) + S(n-1,k-1)$, for $n \geq k+1$ and $k \geq 1$, with the initial conditions $S(n,0) = 1$ and $S(n,n) = 2^n$, for $n \geq 0$. The assertion can now be proved by an easy induction. □

We denote by $\mathcal{S}$ the class of complete solvers with the following three properties: (1) they compute answer sets (or determine that no answer set exists) by generating a sequence of partial assignments so that if an answer set exists then it occurs among the generated assignments; (2) they use boolean constraint propagation to force truth assignments on unassigned atoms and trigger backtracking if contradictions are found; and (3) the generated assignments can be represented by a binary tree, whose nodes are atoms, and where the left (right) edge leaving an atom corresponds to assigning that atom *false* (*true*). This class of solvers includes in particular solvers that use chronological backtracking, as well as those that perform backjumping (we note that in the latter case, some nodes corresponding to decision atoms may have only one child).

Proposition 5. *Let $P \in R_n^-$ be such that the maximum size of an independent set in $G(P)$ equals β. Then, the number of assignments generated by any solver in the class $\mathcal{S}$ is $O((2n)^{\beta+1})$.*

Proof. The tree representing the space of assignments generated by a solver from $\mathcal{S}$ for P has height at most n and at most $\beta + 1$ left edges on every path. Indeed, if there are ever $\beta + 1$ left edges on a path in the tree, then $\beta + 1$ atoms are set to false. Atoms in that set

do not form an independent set in $G(P)$, and so for some two of them, say a and b, the rule $a \leftarrow not\ b$ is in P. Boolean propagation forces a or b to be true, while both of these atoms are false. Thus, a backtrack will occur (the current path will not be extended). The assertion follows now by Proposition 4, as $\binom{n}{k} \leq n^k$. □

We noted earlier that when $m \geq cn^2$, $\beta = O(\log n)$. Thus, when $m \geq cn^2$, the size of the search space is bounded by $n^{O(1)}2^{O(\log^2 n)}$, which is asymptotically much smaller than $O(2^n)$. Furthermore, with m getting closer to $n(n-1)$, β gets even smaller, and so the search space gets smaller, too.

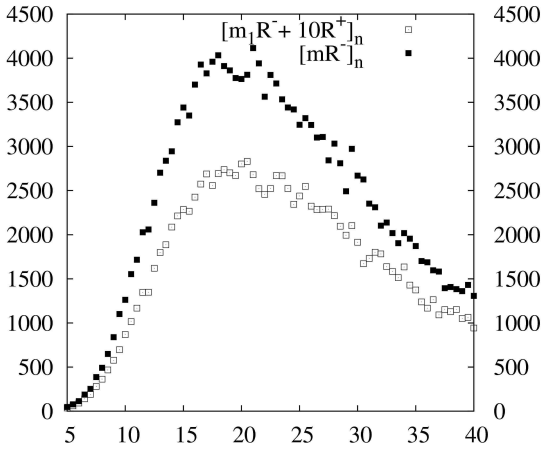


Figure 3.5: Average number of choice points for consistent programs with 150 atoms for *clasp*. The x-axis represents the density. Sample size is 100 consistent programs.

Finally, we note that adding even a small number of positive rules or constraints to programs from mR_n^- generally makes the resulting programs easier. For instance, adding 10 random positive rules to programs from mR_n^-, where $n = 150$ results in about 31% drop in the average number of choice points for *clasp* at the peak location for unsatisfiable instances as shown here in Figure 3.6, and about 32% drop in the average choice points for *clasp* at the peak region for satisfiable instances as shown in Figure 3.5.

On the contrary, adding 100 constraint rules of the form $\leftarrow not(a), not(b)$ to programs from mR_n^- results in a smaller drop of 15% in the average number of choice points for unsatisfiable instances in the peak region as shown here in Figure 3.7, and an increase of 18% in the number of choice points on satisfiable instances in the peak region as shown here

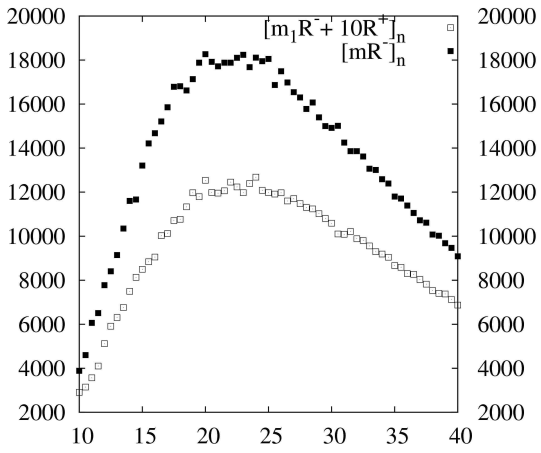


Figure 3.6: Average number of choice points for inconsistent programs with 150 atoms for *clasp*. The x-axis represents the density. Sample size is 100 inconsistent programs.

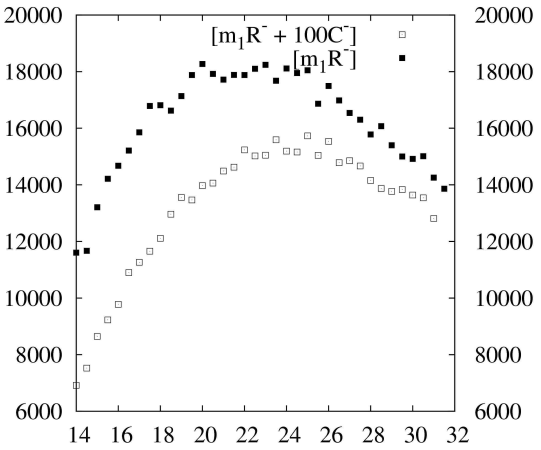


Figure 3.7: Average number of choice points for inconsistent programs with 150 atoms for *clasp*. The x-axis represents the density. Sample size is 100 inconsistent programs.

in Figure 3.8. We found that adding 100 constraint rules of the form $\leftarrow a, b$ to programs from mR_n^- with $n = 150$ results in only trivially unsatisfiable instances for *clasp*. We also noted on adding 100 constraint rules of the form $\leftarrow not(a), b$ to programs from mR_n^- results in mostly unsatisfiable instances and a very significant drop of more than 80% in the average number of choice points on all instances for *clasp*.

These results suggest that from the perspective of benchmarking and insights into search heuristics, proper purely negative constraint-free programs are especially important.

We have shown in this chapter that relatively small programs from the hard region are

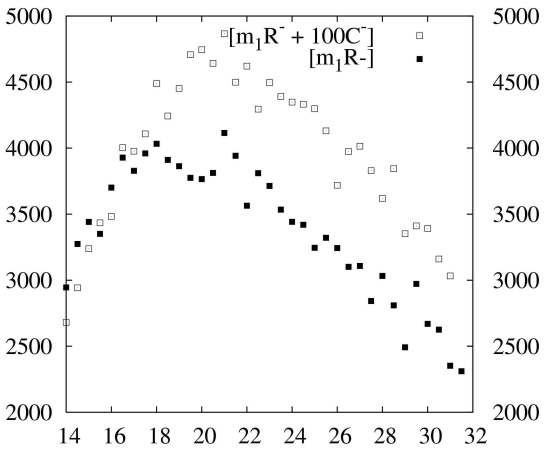


Figure 3.8: Average number of choice points for consistent programs with 150 atoms and *clasp*. The x-axis represents the density. Sample size is 100 consistent programs.

very hard for the current generation of ASP solvers. Interestingly, this observation has implications for the design of SAT solvers, since the completion of constraint-free and purely negative 2-literal programs is (essentially) a CNF theory which is also a mixed Horn formula. We consider in our next Chapter 4 such MHFs with additional restriction on their structure.

Chapter 4

Mixed Horn Formulas

In the recent past there has been significant progress in the study of mixed Horn formulas (MHFs), such as: showing that the satisfiability of these formulas is NP-complete [63], demonstrating that several NP-complete problems have simple representations as MHFs [63], and also developing satisfiability algorithms for MHFs with good worst-case behavior upper bounds [46, 64].

In this chapter we define a few restricted classes of MHFs. Our motivation to consider these special classes of MHFs comes from our work on random logic programs which we discussed in Chapter 3. We had shown in Chapter 3 that purely negative constraint-free 2-literal logic programs from the class mR^- are hard for ASP solvers. The completion of a program from this class results in a CNF formula which is a MHF that consist of purely positive (i.e., containing only non-negated variables) 2-literal clauses and purely negative Horn clauses (i.e., containing only negated variables). We define in this chapter classes of such specially structured MHFs with several constraints imposed on them. We show that the problem of deciding the satisfiability of a formula from the classes of MHFs considered by us remains NP-complete. We also provide a method for randomly generating a formula from any of these MHF classes. Despite the simplicity in the structure of these special classes of MHFs, we show that we can randomly generate very hard formulas for existing SAT solvers from them.

In Section 4.1 of this chapter we define four classes of MHFs. In Section 4.2 we provide a method for randomly generating a MHF from any of these classes. In Section 4.3 we provide our experimental results, which show a phase-transition for the probability of existence of an answer set as we randomly generate formulas from one particular class of MHFs. Section 4.4, Section 4.5, and Section 4.6 demonstrate the hardness of these formulas for SAT solvers based on the average number choice points generated by a solver on

computing the satisfiability of these MHFs. Finally, in Section 4.7 we show that these random MHFs can be used as hard benchmark problems for testing the performance of SAT solvers.

Our work in this chapter appears in Proceedings of SAT-2010 [60].

4.1 Preliminaries

Let $\mathcal{V} = \{v_1, v_2, \ldots\}$ be a fixed set of propositional variables. We define the class $MH_n(k, m)$, where $k \geq 1$ and $m \geq 0$ are integers, to consist of a MHF F such that

1. the set of atoms occurring in F is $\{v_1, \ldots, v_n\}$,
2. F contains m positive 2-clauses,
3. for every $v \in \mathcal{V}$, F contains a negative clause $C_v = \neg v \vee \neg w_1 \vee \ldots \vee \neg w_k$, where $w_1, \ldots w_k \in \mathcal{V}$ (note: clauses C_v and C_w, $v \neq w$, need not be distinct), and
4. there are no other clauses in F.

We also define $MH_n(k) = \bigcup_m MH_n(k, m)$ (here m ranges from 0 to $\binom{n}{2}$), and $MH(k) = \bigcup_{n=0}^{\infty} MH_n(k)$.

Example 32. *We provide here a formula F from the class $MH_n(k, m)$ where $k = 2$, $m = 4$, and $n = 5$.*

$$F = \{v_1 \vee v_2,$$
$$v_3 \vee v_4,$$
$$v_2 \vee v_3,$$
$$v_1 \vee v_5,$$
$$\neg v_1 \vee \neg v_3 \vee \neg v_4,$$
$$\neg v_2 \vee \neg v_5 \vee \neg v_4,$$
$$\neg v_3 \vee \neg v_1 \vee \neg v_5,$$
$$\neg v_4 \vee \neg v_2 \vee \neg v_3,$$

$$\neg v_5 \vee \neg v_1 \vee \neg v_2\}.$$

$\triangle$

We consider also a more general class of MHFs than $MH_n(k,m)$. We denote it by $CMH_n(k,m)$. The parameters n and m in $CMH_n(k,m)$ are the same as in $MH_n(k,m)$, while k is a real number. The class $CMH_n(k,m)$ allows for more gradual changes in the structure of formulas as k grows compared to the class $MH_n(k,m)$ with only integer values. The n purely negative Horn clauses are either of length $\lfloor k \rfloor + 1$ or of length $\lfloor k \rfloor + 2$. The fractional part of k is used to determine the number $c_{\lfloor k \rfloor + 2}$ of purely negative Horn clauses that have $\lfloor k \rfloor + 2$ literals, which is,

$$c_{\lfloor k \rfloor + 2} = \lfloor k * n \rfloor - (n * \lfloor k \rfloor).$$

The remaining Horn clauses have $\lfloor k \rfloor + 1$ literals. The number of those clauses is given by

$$c_{\lfloor k \rfloor + 1} = n - c_{\lfloor k \rfloor + 2}$$

We would like to note that if F is a formula in $CMH_n(k,m)$, where k is an integer, then each of the n Horn clauses has exactly $k+1$ negated literals (i.e., $c_{\lfloor k \rfloor + 2} = 0$, and $c_{\lfloor k \rfloor + 1} = n$).

We formally define the class $CMH_n(k,m)$, where $k > 1$ and $m \geq 0$, to consist of a MHF F such that

1. the set of atoms occurring in F is $\{v_1, \ldots, v_n\}$,
2. F contains m positive 2-clauses,
3. for every $v \in \mathcal{V}$, F contains either a negative clause $C_v = \neg v \vee \neg w_1 \vee \ldots \vee \neg w_{\lfloor k \rfloor}$ of length $\lfloor k \rfloor + 1$, or a negative clause of the form $C_v = \neg v \vee \neg w_1 \vee \ldots \vee \neg w_{\lfloor k \rfloor} \vee \neg w_{\lfloor k \rfloor + 1}$ of length $\lfloor k \rfloor + 2$ where $w_1, \ldots w_{\lfloor k \rfloor}, w_{\lfloor k \rfloor + 1} \in \mathcal{V}$,
4. F has exactly $c_{\lfloor k \rfloor + 2}$ negative clauses of length $\lfloor k \rfloor + 2$, and $c_{\lfloor k \rfloor + 1}$ negative clauses of length $\lfloor k \rfloor + 1$, and

5. there are no other clauses in F (note: we can have two clauses C_v and C_w that are the same)

We also define $CMH_n(k) = \bigcup_m CMH_n(k,m)$ (here m ranges from 0 to $\binom{n}{2}$), and $CMH(k) = \bigcup_{n=0}^{\infty} CMH_n(k)$. Here is an example of a formula from the class $CMH_n(k,m)$.

Example 33. *We provide here a formula F from the class $CMH_n(k,m)$ where $k = 1.5$, $m = 4$, and $n = 6$. Then we have,*

$c_{\lfloor 1.5 \rfloor + 2} = \lfloor 1.5 * 6 \rfloor - 6 * \lfloor 1.5 \rfloor => c_3 = 9 - 6 = 3$, *and*

$c_{\lfloor 1.5 \rfloor + 1} = 6 - c_{\lfloor 1.5 \rfloor + 2} => c_2 = 6 - 3 = 3$.

$$F = \{v_2 \vee v_4,$$

$$v_3 \vee v_5,$$

$$v_6 \vee v_1,$$

$$v_4 \vee v_5,$$

$$\neg v_1 \vee \neg v_2,$$

$$\neg v_2 \vee \neg v_4,$$

$$\neg v_3 \vee \neg v_5,$$

$$\neg v_4 \vee \neg v_1 \vee \neg v_3,$$

$$\neg v_5 \vee \neg v_2 \vee \neg v_6,$$

$$\neg v_6 \vee \neg v_1 \vee \neg v_3\}.$$

$\triangle$

We consider yet another class $MH_n^1(k)$, which we define as follows: an MHF $F \in MH_n(k)$ belongs to $MH_n^1(k)$ if its set of positive 2-clauses is given by

$$\{v \vee w \mid w \in Var(C_v), \text{where } C_v \in F\}.$$

In the case of MHFs from the class $MH_n^1(k)$, there is a strong connection between the sets of positive and negative clauses: if $F \in MH_n^1(k)$, then F is *entirely* determined by its

negative part. We note that the number of 2-clauses in formulas in $MH_n^1(k)$ is not fixed. Each formula from the class $MH_n^1(k)$ can have kn such clauses by definition. However, since each of the 2-literal clause can appear twice in any formula (i.e., if $v_1 \vee v_2$ is such a clause then there is a possibility of having $v_2 \in Var(C_{v_1})$ and $v_1 \in Var(C_{v_2})$), each formula can actually have only $kn/2$ such 2-clauses. An example of a formula from the class $MH_n^1(k)$ is provided here.

Example 34. *We provide here a formula F from the class $MH_n^1(k)$ where $k = 2$, and $n = 5$.*

$$F = \{v_1 \vee v_4,$$

$$v_1 \vee v_5,$$

$$v_2 \vee v_3,$$

$$v_2 \vee v_4,$$

$$v_3 \vee v_2,$$

$$v_3 \vee v_1,$$

$$v_4 \vee v_5,$$

$$v_4 \vee v_3,$$

$$v_5 \vee v_4,$$

$$v_5 \vee v_2,$$

$$\neg v_1 \vee \neg v_4 \vee \neg v_5,$$

$$\neg v_2 \vee \neg v_3 \vee \neg v_4,$$

$$\neg v_3 \vee \neg v_2 \vee \neg v_1,$$

$$\neg v_4 \vee \neg v_5 \vee \neg v_3,$$

$$\neg v_5 \vee \neg v_4 \vee \neg v_2\}.$$

△

Similarly, we define the class $CMH^1_n(k)$ as follows: an MHF $F \in CMH_n(k)$ belongs to $CMH^1_n(k)$ if and only if its set of positive 2-clauses is given by

$$\{v \vee w \mid w \in Var(C_v), \text{where } C_v \in F\}.$$

Here k being real allows us to generate programs from the class $CMH^1_n(k)$ with a gradually increasing k which implies that we can generate formulas with a gradually increasing number of 2-literal clauses.

We write $CMH^1(k)$ for $\bigcup_{n=0}^{\infty} CMH^1_n(k)$, and CMH^1_n for $\bigcup_{k=1} CMH^1_n(k)$. Despite constraints on the form of MHFs that form the classes $CMH(k)$ and $CMH^1(k)$, for each of them the satisfiability remains NP-complete.

4.2 Method for the generation of MHFs

We provide here the method we use for the generation of formulas from the classes $CMH_n(k,m)$ and $CMH^1_n(k)$. Since the class $CMH_n(k,m)$ is a more general class of formulas than $MH_n(k,m)$, we discuss here only the method used to generate formulas from $CMH_n(k,m)$. Similarly, since the class $CMH^1_n(k)$ is a more general class of formulas than $MH^1_n(k)$, we discuss here only the method used to generate formulas from $CMH^1_n(k)$.

We generate a formula from the class $CMH_n(k,m)$ in the following way:

1. For each variable $v_i \in V$, where $1 \leq i \leq c_{\lfloor k \rfloor+1}$, we initially generate a set V_S of $\lfloor k \rfloor$ variables by choosing them uniformly at random from $\mathcal{V} \setminus \{v_i\}$. We then construct a clause that contains $\neg v_i$ and the negation of each variable in the set V_S.

2. For each variable $v_j \in V$, where $(c_{\lfloor k \rfloor+1}+1) \leq j \leq c_{\lfloor k \rfloor+2}$, we generate a set V_S of $\lfloor k \rfloor+1$ variables by choosing them uniformly at random from $\mathcal{V} \setminus \{v_i\}$. We construct a clause that contains $\neg v_j$ and the negation of each variable V_S in the set.

3. Each of the positive 2-clauses is obtained by uniformly and randomly choosing m clauses from the set of all possible $\binom{n}{2}$ such clauses without replacement.

The negative clauses for a formula from the class $CMH^1_n(k)$ are generated in the same manner as in the case of a formula from the class $CMH_n(k)$. However, positive 2-clauses

for a formula from the class $CMH^1_n(k)$ are directly constructed from the negative clauses just as provided in its class definition.

Proposition 6. *For each of the classes $CMH(k)$, $CMH^1(k)$, $MH^1(k)$, and $MH(k)$ with $k \geq 2$, the satisfiability problem restricted to that class of formulas is NP-complete.*

Proof. In each case the problem is in NP. To prove NP-hardness, we note that the classes $CMH(k)$, $CMH^1(k)$, $MH(k)$ represent a more general class of formulas than $MH^1(k)$. Hence it is sufficient to show the NP-hardness for $MH^1(k)$, which we show here by providing a polynomial time reduction from a simple class of logic programs [55] consisting of rules of the form $a \leftarrow not(b)$ to the class of $MH^1(k)$. We will consider only the case $k = 2$.

Let P be a program whose every rule is of the form $a \leftarrow not(b)$. Let us assume that P contains rules $a \leftarrow not(b)$, $a \leftarrow not(c)$, and $a \leftarrow not(d)$, for some three different atoms b, c, and d. Let Q be the program obtained by replacing two of the three rules: $a \leftarrow not(c)$ and $a \leftarrow not(d)$ in P with the rules

$a \leftarrow not(a')$

$a' \leftarrow not(a'')$

$a'' \leftarrow not(c)$

$a'' \leftarrow not(d)$

where a' and a'' are two new atoms. Proceeding in this way we construct program Q such that every atom is in the head of at most two rules.

We will now show that P has an answer set if and only if Q has an answer set. Since both P and Q are tight, it is enough to show that P has a supported model if and only if Q has a supported model.

Let M be a supported model of P. We define M' as follows:

1. If $a \in M$, $c \notin M$ or $d \notin M$, then we let $M' = M \cup \{a''\}$,
2. If $a \in M$, $c \in M$ and $d \in M$, then we let $M' = M \cup \{a'\}$;
3. If $a \notin M$ then $M' = M \cup \{a'\}$.

We will show that M' is a model of Q and that every atom in M' is supported w.r.t. Q and M. Since M is a model of P, M' is a model of every rule in Q that also belongs to P. Thus, let r be a rule in $Q \setminus P$. Then, r is one of the four rules given above. In each case, we can check that M' is a model of r. For instance, let $r = a \leftarrow not(a')$. If $a' \notin M'$ then, by the definition of M', $a \in M$ and so $a \in M'$. Thus, M' is a model of r.

Next, let $x \in M'$. We will show that x is supported w.r.t. Q and M'. If $x \in At(P)$ and $x \neq a$, then since M is a supported model of P, there is a rule r in P that gives support to x, say $r = x \leftarrow not(y)$, where $y \notin M$. Clearly, $r \in Q$ and, by the construction of M', $y \notin M'$. Thus, x has support w.r.t. Q and M'. If $x \in At(P)$ and $x = a$, then $a \in M$ and since M is a supported model of P, there exists a rule $r \in P$ such that $r = a \leftarrow not(z)$, for some $z \in At(P)$, and $z \notin M$. If $r \in Q$, then it must be the case that $z \notin M'$ (as $z \in At(P)$ and $z \in M$). Thus x is supported w.r.t Q and M'. If $r \notin Q$ then $r = a \leftarrow not(c)$ and $r = a \leftarrow not(d)$. They have been replaced in Q with the four rules as described above. If $r = a \leftarrow not(c)$, then $c \notin M$, and by definition of M', $a' \notin M'$, and so a is supported by a rule $a \leftarrow not(a')$ in Q. Hence, x has support w.r.t. Q and M'. Similarly, if $r = a \leftarrow not(d)$, then by definition of M' if $d \notin M$, $a' \notin M'$ and so a is supported by a rule $a \leftarrow not(a')$ in Q. Hence, X has support w.r.t. Q and M'.

If $x \notin At(P)$ then $x \in \{a', a''\}$. Let us assume that $x = a'$. Then, there is exactly a single rule $r = a' \leftarrow not(a'')$ where $head(r) = a'$ in Q, and by definition of M' if $a' \in M'$ then, $a'' \notin M'$. Hence, x is supported w.r.t. Q and M'. If $x = a''$, then there are two rules $r = a'' \leftarrow not(c)$ and $r = a'' \leftarrow not(d)$ in Q with $head(r) = a''$, and by definition of M', if $a'' \in M'$, then either $c \notin M'$ or $d \notin M'$. Hence, x is supported w.r.t. Q and M'.

Next, we prove the converse implication. Let M' be a supported model of Q. Let us define $M = M' \cap At(P)$. We will show that M is a supported model of P. We will show that M is a model of P and that every atom in M is supported w.r.t. P and M. Since M' is a model of Q, M is a model of every rule in P that belongs to Q. Thus, let r be a rule in $P \setminus Q$. Then, r is either $a \leftarrow not(c)$ or $a \leftarrow not(d)$, and these have been replaced in Q with the four other rules $a \leftarrow not(a')$, $a' \leftarrow not(a'')$, $a'' \leftarrow not(c)$, and $a'' \leftarrow not(d)$ in Q. In each case we can check that M is a model of r. For instance, let $r = a \leftarrow not(c)$.

If $c \notin M$, then $c \notin M'$. Since M' is a supported model of the four other rules, $a'' \in M'$, $a' \notin M'$, and $a \in M'$. If $a \in M'$, then $a \in M$. Thus, M is a model r.

Next, let $x \in M$. We will show that x is supported w.r.t. P and M. Since $x \in M$, $x \in M'$. Then, x is supported by a rule r in Q. Let us assume that $r \in Q \cap P$. If $r = x \leftarrow not(y)$, then $y \notin M'$. If $y \notin M'$, $y \notin M$. Hence, x has support w.r.t. P and M. Let us then assume that $r \in Q \setminus P$. Then $x = a$ and r must be the rule $a \leftarrow not(a')$, since a has support w.r.t. Q and M'. Thus, $a' \notin M'$. Then $a'' \in M'$. If $a'' \in M'$ then $c \notin M'$ or $d \notin M'$. Let us assume that $c \notin M'$. Then $c \notin M$. Thus x has support w.r.t. P and M by means of the rule $a \leftarrow not(c)$. Similarly, if we assume $d \notin M'$, $d \notin M$. Then x must be supported w.r.t. P and M by the rule $a \leftarrow not(d)$.

Clearly, repeating the replacement process as long as needed, we will construct a program Q that has exactly the same answer sets as P and in which no atom shows up as the head of more than two rules.

Next, let us assume that in this Q there is a program in which an atom a is the head of one rule $a \leftarrow not(b)$ only. Let us append it with the rules

$a \leftarrow not(x_0)$

$x_0 \leftarrow not(x_1)$

$x_0 \leftarrow not(x_2)$

$x_1 \leftarrow not(x_2)$

$x_2 \leftarrow not(x_3)$

$x_3 \leftarrow not(x_4)$

$x_4 \leftarrow not(x_1)$

$x_2 \leftarrow not(x_1)$

$x_3 \leftarrow not(x_2)$

$x_4 \leftarrow not(x_3)$

$x_1 \leftarrow not(x_4)$

where $x_0, \ldots, x_4$ are new atoms. Let us call the resulting program R. We will show that Q has an answer set if and only if R has an answer set. We will proceed as before and show the equivalence for supported models as the programs are tight.

Let M be a supported model of Q. We define $M' = M \cup \{x_0, x_1, x_3\}$. We will show that M' is a model of R, and is supported w.r.t R and M'. Since, M is a model of Q, M' is a model of every rule in R that belongs to Q. Thus, let r be a rule in $R \setminus Q$. Then, r is one of the eleven rules given above. In each case we can check that M' is a model of r. Thus, M' is a model of R.

Next, let $x \in M'$. We show that x is supported w.r.t. R and M'. If $x \in At(P)$, and $x \neq a$, then since M is a supported model of Q, there is a rule r in Q that gives support to x, say $r = x \leftarrow not(y)$, where $y \notin M$. Clearly, $r \in R$, and, by the construction of M', $y \notin M'$. Thus x has support w.r.t. R and M'. If $x \in At(P)$, and $x = a$, then since M is a supported model of Q, there is a single rule $r = a \leftarrow not(b)$ where $b \notin M$, that gives support to x. Clearly, $r \in R$, and, by the construction of M', $b \notin M'$. Thus x has support w.r.t. R and M'. If $x \notin At(P)$, then $x \in \{x_0, x_1, x_3\}$. In each case we can check that x is supported w.r.t. R and M'. For instance, if $x = x_0$, then it is supported by the rule $x_0 \leftarrow not(x_2)$. Hence, x is supported w.r.t. R and M'.

Next, we prove the converse implication. Let M' be a supported model of R. Let us define $M = M' \cap At(Q)$. We will show that M is a supported model of Q.

We will show that M is a model of Q, and is supported w.r.t Q and M. Since, M' is a model of R, M is a model of every rule in Q that belongs to R. Thus M is a model of Q (since $Q \subseteq R$).

Next, let $x \in M$. We will show that x is supported w.r.t. Q and M. Let $x \neq a$. Since M' is a supported model of R and $x \in M'$, there is a rule r in R that gives support to x, say $x \leftarrow not(y)$, where $y \notin M'$. Clearly, r in Q, by the construction of M, and $y \notin M$. Thus, x has support w.r.t. Q and M. If $x = a$, then there are two rules $r \in R$ that can provide support to a. They are $a \leftarrow not(b)$, and $a \leftarrow not(x_0)$. However, one can check that since M' is a supported model of R, $x_0 \in M'$. So x must be supported only by the rule $r = a \leftarrow not(b)$. Then $b \notin M'$. Clearly, $r \in Q$, and, by the construction of M, $b \notin M$. Thus x has support w.r.t. Q and M.

Clearly, repeating the replacement process as long as needed, we will construct a program R that has exactly the same answer sets as Q and in which every atom is in the head

of exactly two rules.

We have thus shown here that the resulting program R has the same answer sets as that of P. It follows that P has answer sets if and only if its completion R_{comp} has answer sets. Moreover, every atom is in the head of exactly two rules. Hence, $R_{comp} \in MH^1(2)$. Thus the NP-hardness of the SAT problem for $MH^1(2)$ follows.

□

4.3 Phase transition

We randomly generate formulas from the class $CMH_n(k)$ for a fixed k and n, but with an increasing number of 2-literal positive clauses m. We experimentally compute the probability of existence of a model for formulas randomly generated from this class, and observe a phase transition for the probability of existence of a model for formulas generated from this class with increasing density (i.e., ratio of m to n) of 2-literal rules. The probability of existence of a model is initially 1 (i.e., satisfiable) for small m, and then shows a sharp transition to 0 (i.e., unsatisfiable) with increasing m.

We show here in Figure 4.1 and Figure 4.2 the phase transition phenomenon for $k = 5$, $k = 10$ with $n = 50, 100, 150, 200$, and $n = 250$. The threshold gets sharper with increasing n and approximately coincides at the same critical region denoted by c_k (i.e., the value for m where the probability of existence of the model is 0.5). The value of m at c_k for $k = 5$ and $k = 10$ is approximately $4.3n$ and $9.4n$.

Here, the phase transition phenomenon is not surprising due to the following reasons: every formula that we generate from the class $CMH_n(k)$ has always the same fixed number of negative Horn clauses; initially we generate formulas that have very few positive 2-literal clauses m and are under-constrained; and as we generate formulas with increasing m these formulas gradually become more and more constrained.

The approximate location of the phase-transition region expressed in terms of the density m/n, for which the phase transition occurs, grows with k as we randomly generate formulas from the class $MH_n(k)$ with increasing density of 2-literal rules. Our experimental results for $n = 200$ and $k = 3, \ldots, 25$ (200 instances) show that the location of the

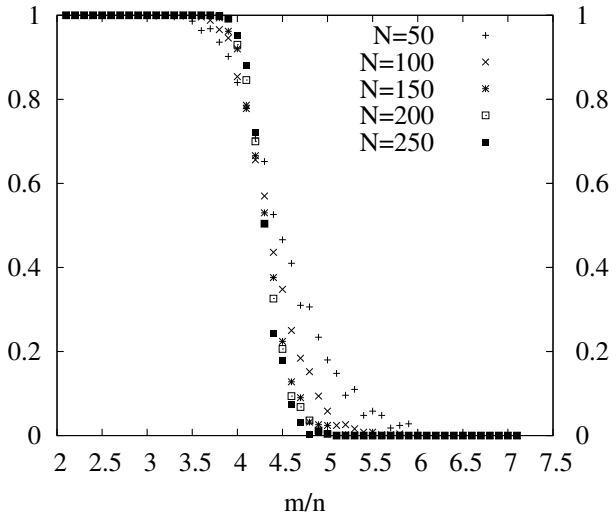


Figure 4.1: The phase transition for the model $CMH_n(5)$. The x-axis represents the probability of existence of a model, and the y-axis represents the density of 2-literal rules.

phase transition grows slightly slower than k as seen in Figure 4.3.

4.4 Easy-hard-easy pattern I

The number of choice points generated by a solver, to find a model of a formula, provides an estimate of the size of search space traversed by it. Hence, we use the number of choice points as a parameter in addition to the time taken by the solver, to measure the difficulty of a randomly generated formula. We compute the average choice points as well as the average time taken by the solver *clasp* on computing the satisfiability of randomly generated formulas from the class $CMH_n(k)$ for a fixed k and n. We compute the average time taken and the average choice points over all satisfiable and unsatisfiable formulas. We observe that formulas that are generated with smaller m, much before the critical region c_k, are initially easier for *clasp* requiring less time and fewer choice points. The formulas continue to get harder for *clasp* with increasing m until the density of the 2-literal clauses reaches a peak value m_k (corresponding to the density c_k at the critical region). Then the formulas start to get easier for *clasp* as we increase the density of 2-literal clauses past the critical region.

Hence, we observe an easy-hard-easy pattern as we compute the average time and the

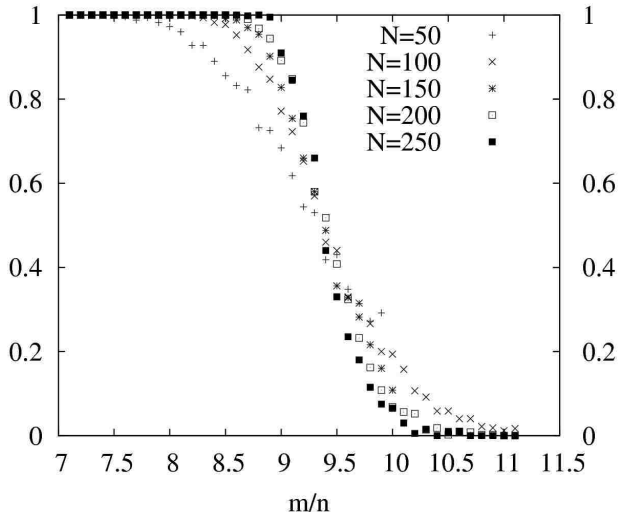


Figure 4.2: The phase transition for the model $CMH_n(10)$. The x-axis represents the probability of existence of a model, and the y-axis represents the density of 2-literal rules.

average choice points made by the solver *clasp* on testing the satisfiability of formulas generated from the class $CMH_n(k)$ for a fixed k and n, and an increasing m. The easy-hard-easy pattern is associated with the phase transition, and the peak hardness for the solver *clasp* coincides with the critical region. We observe this pattern in the graph shown in Figure 4.4, for instances generated from the class $CMH_n(10)$ with $n = 150$, corresponding to the phase-transition. We also observed a similar easy-hard-easy pattern for formulas generated from the class with $CMH_n(10)$ and $n = 200$, as well as for formulas generated from the class $CMH_n(5)$ with $n = 150$, and $n = 200$.

We also computed the average choice points made by *clasp* on 500 randomly generated instances provided in Table 4.1 from each of the classes: $CMH_n(k)$ where $k = \{5, 10\}$, and $n = \{50, 100, 150, 200, 250\}$.

We observe that for a fixed k, the hardness (i.e., the average number of choice points generated by clasp) grows at c_k as we increase n, as shown in Table 4.1. The unsatisfiable formulas generated in the critical region are much harder than the satisfiable ones similar to the random 3-SAT formulas that are generated from the critical region.

We provide in Appendix B further experimental results using the SAT solver *Minisat* (MiniSat_v1.14) [24] on instances obtained from the class $MH^n(k)$. We observe a similar

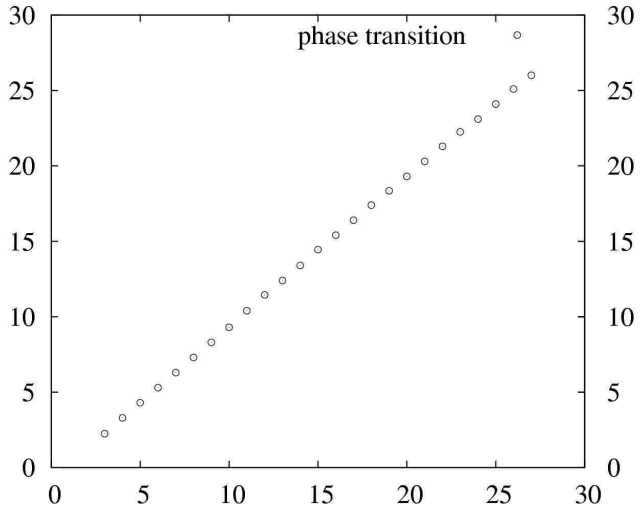


Figure 4.3: The location of the phase transition in the model $MH_n(k)$ as a function of k. The x-axis represents k and the y-axis gives the approximate density of 2-literal rules near the phase transition.

Table 4.1: The average choice points made by *clasp* at the critical region for the model $CMH_n(k)$

	Average choice points (*clasp*)	
n	c_5	c_{10}
50	33.55	70.61
100	254.48	1272.15
150	1688.98	17032.85
200	10016.07	179770.60
250	55966.71	1882953.466

phase-transition and a corresponding easy-hard-easy pattern as we plot the average number of choice points generated by *Minisat* on randomly generated instances from the class $MH^n(k)$, where $n = 200$, $k = 10, 20, 30, 40$, and as m grows.

4.5 Easy-hard-easy pattern II

However, the framework of the classes $CMH_n(k)$ we consider reveals yet another interesting phenomenon. We fix n, and plot the average number of choice points generated by a SAT solver on instances generated from the model $CMH_n(k)$ at the approximate location of the critical region c_k, for increasing values of k. Being parameterized with k, it allows us to compare the hardness of instances generated from the critical region for *different* values

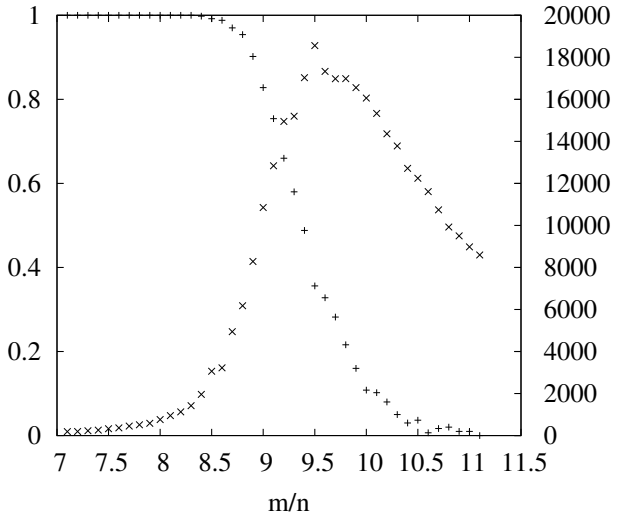


Figure 4.4: The phase transition for the model $CMH_n(10)$ with $n = 150$

of k. Somewhat surprisingly, it turns out that as we increase k, the easy-hard-easy pattern emerges again. Initially, as k grows, the phase-transition instances are getting harder at an increasing rate. The hardness peaks when $k \approx 15, 16$, and from that point on the instances become increasingly easier. Figure 4.5 illustrates that pattern observed for *clasp* for $n = 200$, and k ranging from 3 to 34.

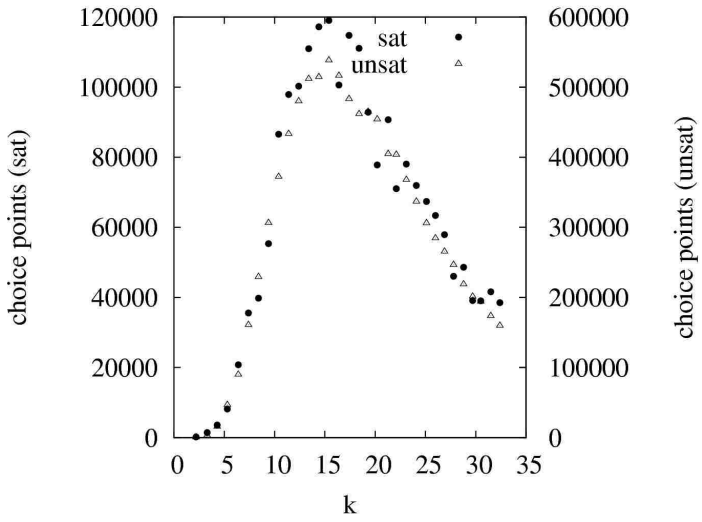


Figure 4.5: The easy-hard-easy pattern of instances generated from the critical region for $MH_n(k)$ as a function of k.

We observe that the hardness of instances from the critical region in the model $CMH_n(k)$

initially grows with k; it may seem surprising that at some point it peaks and then starts to decrease.

4.6 Easy-hard-easy pattern III

We also observe an easy-hard-easy pattern as we generate random instances from the class $CMH_n^1(k)$ with a fixed n, and as k grows. However, we do not observe a phase transition corresponding to the easy-hard-easy pattern, unlike the one observed as we generated formulas from the class $CMH_n(k)$ as m grows. In fact, the probability of existence of a model for formulas generated from the class $CMH_n^1(k)$ is initially 1 for $k = 0$ and rapidly drops close to 0.2 and then gradually rises and reaches closer to 1 again. We generate random instances from the class $CMH_n^1(k)$ with $n = 200$ and increasing k. We observe again an easy-hard-easy pattern for *clasp* as shown here in Figure 4.6, with peak hardness around $k = 15$. The probability of existence of a model for randomly generated formulas from the class $CMH_n^1(k)$ with increasing k is also shown in Figure 4.6.

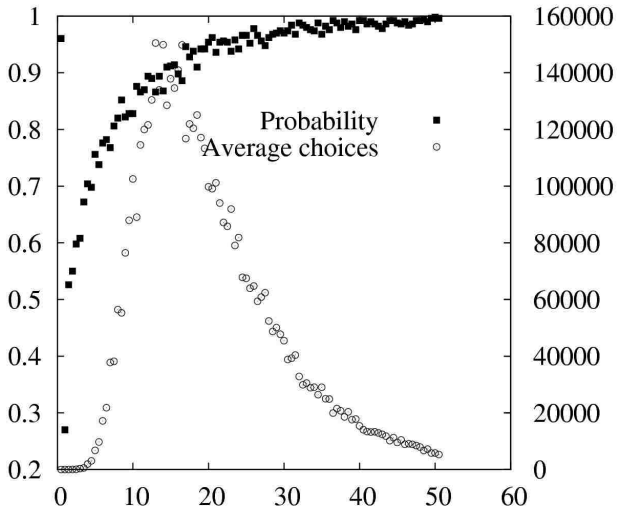


Figure 4.6: The easy-hard-easy pattern for the model $CMH_n^1(k)$, and the probability of satisfiability. The left x-axis represents the probability of existence of a model, the right x-axis represents the average choice points made by *clasp*.

4.7 Hard Benchmarks for SAT Solvers

Our results suggest that MHFs randomly generated from the phase transition region for the class $MH_n(k)$ for $k = 15$ or 16 (located when $m \approx (k - 0.5)n$, where m stands for the number of 2-clauses) can provide challenging instances for SAT solvers. It is indeed so. We randomly generated 50 instances from $MH_n(k, m)$, with $n = 350$, $k = 15$ and $m = 14.5n$. Given the timeout limit of 1800 seconds, *clasp* and *march_hi* solved fewer than 20% of the instances all of which were satisfiable and did not solve any of the unsatisfiable ones.

We stress that the instances in $MH_n(15, 14.5n)$ are small (350 atoms and 5425 clauses), and more important, that most of their clauses (5025) are 2-clauses. Since they pose a challenge for the state-of-the-art complete solvers, we believe that the class $MH_n(15, 14.5n)$ is important for the design and testing of solver performance.

The classes $MH^1_n(k)$ offer even harder instances. While they can also serve as benchmarks for complete solvers, even for relatively small values of n, satisfiable instances from $MH^1_n(15)$ become very hard also for local-search solvers! The selection of $k = 15$ is not accidental. Our experiments showed that when we vary k, we observe the easy-hard-easy pattern, with the peak for $k \approx 15$. We also found that in the maximum hardness area, the percentage of instances that are satisfiable exceeds 90% for different N values, as shown in Figure 4.6.

We generated 100 random CNF formulas from each of the sets $MH^1_n(15)$, where $n = 450$ and 550. Given our experiments, the expected number of satisfiable instances in these two sets of formulas is at least 90. We ran *TNM* [4] on these formulas. *TNM* is currently one of the best local-search solvers. It won in the random category (satisfiable instances only) at the SAT 2009 competition. The solver does not require any parameters, as it adaptively selects them. We observed that for $n = 450$, *TNM* could still solve 86% of the instances in less than 1800 seconds (yet, likely missing some satisfiable instances). The larger value of n, $n = 550$, resulted in many hard instances. Indeed, for $n = 550$, *TNM* solved only 53 of the 100 instances within 1800 seconds, while we expect about 90 instances to be satisfiable in this sample.

Chapter 5

Related Work

5.1 Random Logic Programs

Random logic programs were initially introduced and studied by Zhao and Lin. Their work on generating random logic programs was motivated by the prior works done on generating hard random SAT instances. In their work [80], they consider logic programs of two kinds, those with a fixed number of literals in the body of the rule (fixed body length model) and those with a varying number of literals in the body (variable body length model). Each rule has an atom in its head. Each logic program has 3 parameters: number of atoms (N), rule density α, which specifies that the number of rules (L) in the program is α times N (i.e.,$\alpha = L/N$), and either a fixed number of literals (K) in the body of the rule or a probability distribution (λ) that specifies the probability of occurrence of a rule with a fixed number of literals in its body.

5.1.1 Properties of Random Logic Programs

The authors [80] use SAT solvers to determine the existence of an answer set on randomly generated logic programs with increasing α and show a transition from satisfiable instances to unsatisfiable instances. The SAT solvers demonstrate an easy-hard-easy pattern on these randomly generated logic programs by taking longer time to determine the existence of an answer set for logic programs that are generated with a certain density α_h and being able to quickly compute the existence of an answer set for all other instances that are generated with $\alpha << \alpha_h$ and $\alpha >> \alpha_h$. Thus showing that a hard region exists for SAT solvers at a particular value of α, for instance $\alpha = \alpha_h$, and there exist easy regions for $\alpha << \alpha_h$ and $\alpha >> \alpha_h$.

5.1.2 Fixed Body Length Model

Here, we will recall Zhao and Lin's fixed body length model. Let $FL^K(N, \alpha)$ denote a class of random logic programs with N atoms and $L = \alpha \times N$ rules, where each rule has a fixed length of K ($2 \leq K \leq N$) literals such that $K \setminus 1$ literals appear in the body of the rule and a single atom occurs in the head of the rule. Let At denote the set of N atoms. The probability space

$$\Omega = \{a \leftarrow b_1, \ldots, b_n, not(c_1), \ldots, not(c_m)$$

$$| \quad a \in At, \quad \{b_1, \ldots, b_n, c_1, \ldots, c_m\} \subseteq At, \quad n + m = K - 1\},$$

consists of the set of all the possible rules with $K - 1$ literals in its body. The procedure below is used to generate a program by allowing each rule in it to be selected from this set with equal probability.

Generation of Random Logic Programs ($\boldsymbol{FL^K(N, \alpha)}$)

The authors [80] generate a random logic program from the class $FL^K(N, \alpha)$ in the following way. Let us use $At_N = \{a_1, \ldots, a_N\}$ to denote the set of N atoms. Each of the L distinct rules is generated by

- randomly choosing an atom a_i for the head of the rule,
- randomly choosing $K \setminus 1$ different atoms from At for the body and negating each with probability 0.5, and
- discarding the rule if it has been previously generated.

Experiments on $\boldsymbol{FL^K(N, \alpha)}$

The rules that are generated in each random logic program in the class $FL^K(N, \alpha)$ using the method described above are distinct. Hence, each logic program that is generated from the class $FL^K(N, \alpha)$ can have at most L_{max} (i.e., $\alpha \times N \leq L_{max}$) number of rules in it. This number is given by,

$$L_{max} = N * 2^{K-1} * C^N_{K-1} [80].$$

Let us use α_{max} to denote the rule density of the logic program with L_{max} rules. The authors [80] prove the following:

- There does not exist an answer set for a logic program in the class $FL^K(N, \alpha_{max})$ (with all possible rules).
- There exists a unique answer set for a logic program in the class $FL^K(N, 0)$ (with no rules) which is the empty set.

In the experiments conducted by the authors [80], they generate random logic programs using the method described above. They generate a class of random logic programs $FL^K(N, \alpha)$ with parameters $N = 150$, $K = 3$, and α in the range from 0.5 to 12 in increments of 0.5.

The time taken by the different solvers (smodels, ASSAT, DLV) on these programs as well as the probability (i.e., experimental probability) of existence of an answer set is plotted in their graph [80]. The probability of existence of an answer set with $N = 150$ drops from 1 to 0 as α increases from 0 to 12. The initial drop in probability for $\alpha < 2$ is steep. Each of the solvers show an easy-hard-easy region when α is in the range from 3 to 8 and with maximum hardness occurring when α is close to 5. In the hard region, the probability of existence of an answer set is in the range from 0.1 to 0.2. This is in contrast to the hard region for randomly generated SAT instances [57] which occurs when the probability of existence of a model is 0.5. The authors do not provide a reason for the appearance of an easy-hard-easy region for the class of random logic programs generated by them. However, they relate the hard region to the low probability area, due to the fact that logic programs are non-monotonic and the appearance of a local contradiction does not indicate that the problem is unsatisfiable. Whereas in SAT the appearance of a local contradiction indicates that the problem is unsatisfiable. All three solvers take on average more time to solve an unsatisfiable instance when compared to the time taken to solve a satisfiable instance in the hard region. The average time taken by DLV on all the instances in the hard region is higher when compared to the time taken by smodels and ASSAT, and the average time taken by smodels is greater than the time taken by ASSAT in the same

region.

5.1.3 Mixed Body Length Model

In the mixed body length model the rules in the program have varying numbers of literals in their body. A probability distribution $\Sigma_{n>0}\lambda(n) = 1$ provides the probability $\lambda(n)$ that a rule with $n-1$ literals in its body and a single literal in the head occurs in a randomly generated logic program. We use $ML^{\lambda}(N, \alpha)$ to denote the class of logic programs in the mixed body length model that have N atoms, rule density α, and have the probability distribution λ.

The authors [80] generate random logic programs from the class $ML^{\lambda}(N, \alpha)$ with the following parameters: $N = 100$, $0.5 \leq \alpha \leq 12$, $\lambda(3) = 0.5$, and $\lambda(4) = 0.5$. A phase transition for the probability of existence of an answer set occurred with increasing α, similar to the one observed in the fixed body length model. The authors also observe an easy-hard-easy pattern as they plot the average time taken by the SAT solvers to determine the existence of an answer set for randomly generated programs from this class with increasing rule density, similar to that observed in the first class of instances. The hard region occurs for this class of instances around $\alpha = 6$.

5.2 Random SAT

We provide here an introduction to the generation of random satisfiability (SAT) instances and their properties, since it has motivated the generation of random logic programs. The main motivation for the generation of random SAT instances is to find hard instances to help with testing and improving the performance of solvers. The initial generation of hard random SAT instances had motivated researchers to understand the properties of these instances and the reasons for their hardness.

There has been research in generating random K-SAT instances in particular random 3-SAT instances and understanding their properties. This is because every SAT instance can be represented as a 3-SAT instance. In the following subsection we discuss a method for generating random K-SAT instances. We also discuss the experimental results obtained

by using the DPLL algorithm (used in the majority of SAT solvers) on these SAT instances as seen in [57].

5.2.1 Generation of Random SAT instances

There are many methods [57] for generating random SAT instances. We discuss one of the methods used for generating random K-SAT instances with a fixed number of literals (K) in each clause. We describe in this section one of the earliest works on generating and observing properties such as the phase transition and the easy-hard-easy region for random SAT instances, which was done by David Mitchell et al. [57].

Let us use $RSAT^K(N, \beta)$ to denote the class of random SAT instances where each instance has a set of atoms N, clause density β, $\beta \times N$ clauses, and every clause has a fixed number of K distinct atoms in it. The set of K atoms in a clause is chosen randomly from the set of atoms N, and each atom is negated with probability 0.5. The maximum number of clauses that a random instance in the class $RSAT^K(N, \beta)$ can have is $2^K \binom{N}{K}$. In this method, each of the $\beta \times N$ clauses is randomly and uniformly chosen without replacement from the set of all $2^K \binom{N}{K}$ clauses.

5.2.2 Properties of $RSAT^K(N, \beta)$ instances

There are four key properties observed in randomly generated $RSAT^K(N, \beta)$ SAT instances using the method described above. They are as follows.

- There is a phase transition from satisfiable instances to unsatisfiable instances (i.e., the probability that the randomly generated instance is satisfiable drops from 1 to 0) as we randomly generate SAT instances from the class $RKSAT^K(N, \beta)$, with increasing clause density β. The region where the transition in the probability takes place (i.e., the probability that the randomly generated instance is satisfiable is strictly less than 1 or is strictly greater than 0) is narrow when compared to the region where the probability that the randomly generated instance is satisfiable is exactly 1 or 0.

- There is a region (specified by β) called the *critical region* where the average number of calls made by the DPLL algorithm on all instances (both satisfiable and un-

satisfiable) is much larger than elsewhere. This region is directly correlated to the *cross-over point* which is the point at which the probability of generating a satisfiable instance is 0.5.

- The cross-over point for a fixed clause length K appears at approximately the same clause density d as N is varied, for instance when $K = 3$ the cross-over point is approximately 4.258 [20].

- There are easy regions on either side of the critical region. Hence, there is an easy-hard-easy pattern observed for $RKSAT^K(N, \beta)$ instances that are generated with increasing β.

Experimental results on $RSAT^K(N, \beta)$

In their work [57] the authors generate $RSAT^3(50, \beta)$ instances for increasing values of β. They observe a phase transition for the probability of existence of a model for $RSAT^3(50, \beta)$ instances that they generate with increasing β. The probability that the randomly generated instance is satisfiable is closer to 1 when $\beta \leq 3$, and this probability converges to 0 when $\beta \geq 6$.

The authors plot the average number of recursive calls to the DPLL algorithm which corresponds to the number of choice points made by the DPLL algorithm on satisfiable, unsatisfiable, and all (both satisfiable and unsatisfiable) $RSAT^3(50, \beta)$ instances [57]. The authors notice that the average number of choice points made by all instances is maximum in the critical region ($\beta = 4.258$). However, on either side of the critical region the average number of choices made by all the instances is much smaller. Hence, an easy-hard-easy pattern was observed for $RSAT^3(50, \beta)$ instances. A similar phase transition and easy-hard-easy pattern for $RSAT^3(N, \beta)$ instances with $N = 20, 40$ was observed by the authors [57]. However, the average number of choice points increases as N increases for $RSAT^3(N, \beta)$ instances that have the same β.

The instances that are generated on the left-hand side of the critical region are under-constrained due to the low clause density, and the DPLL algorithm requires less time to

find a model of the instance. So as the clause density decreases, the probability that the instance being generated is satisfiable increases and reaches 1. On the right-hand side of the critical region the instances have a high clause density and are over-constrained. In the over-constrained region the DPLL algorithm is able to quickly determine that the instance is unsatisfiable. Hence, as the clause density increases, the probability that an instance being generated is satisfiable decreases and reaches 0.

The phase transition and easy-hard-easy pattern was also observed for instances generated in the class $RSAT^K(25, \beta)$ with $K \in \{2, 3, 4, 5\}$ [20].

There has also been work on trying to understand the exact reasons for the occurrence of the hard region in random SAT instances. The recent works include identifying features such as backbones [79], backdoors [45], satisfiable cores [78], and unsatisfiable cores [52] and relating them to the occurrence of the hard region. However, no satisfactory explanations have yet emerged.

5.2.3 Threshold for random SAT

In the early 90's a satisfiability threshold conjecture [16] was proposed. It states that there is a constant β_K called the satisfiability threshold that is dependent on the fixed clause length K such that

$$\lim_{N \to \infty} Pr[RSAT^K(N, \beta) \text{ is satisfiable}] = \begin{cases} 1 & \text{if } \beta < \beta_K \\ 0 & \text{if } \beta > \beta_K \end{cases}$$

The value of β_2 is proven to be 1 [16]. Since the early 90's and for more than a decade much of the research in determining the satisfiability threshold [23, 43] was focussed on trying to determine the lower bound and upper bound for β_3 and for the general case β_K, $K \geq 3$ [8, 6, 29, 31, 76]. The best-known lower bound for β_3 is 3.52 [38] and the best-known upper bound is 4.506 [23]. Moreover, in 2006 the threshold value for β_K for $K \geq 3$ was asymptotically approximated as $\Theta(2^K)$ [7].

Chapter 6

Conclusions

The goal of this thesis is to develop techniques to generate hard random theories and programs that can be challenging for existing solvers, as well as to study the properties of these theories. These hard benchmarks can be used to evaluate the performance of the existing solvers. In the past, the generation of hard random SAT formulas has had a substantial positive effect on the design and performance of SAT solvers. It has also motivated research for more than a decade that focussed on understanding the experimental and theoretical properties of these randomly generated formulas, as well as on improving the efficiency of satisfiability testing algorithms.

Our work described in this thesis, like the work done earlier in the area of random SAT and random logic programs, is aimed at generating hard instances for both ASP and SAT solvers. However, we differ by considering theories and programs with especially simple structure for existing ASP and SAT solvers. Our initial motivation to generate hard random logic programs comes from the work done in [80] and the research done in the area of random SAT [6, 8, 20, 29, 31, 45, 52, 57, 76, 78, 79]. A direct linear-space translation that exists for tight logic programs to CNF theories [28], especially a translation from tight 2-literal purely negative constraint-free logic programs to MHFs, further motivated our interest in generating hard and simple MHFs for SAT solvers.

In this thesis we have considered a model of random logic programs in which every rule has exactly two literals. Our model allows for different combinations of normal rules and constraints of particular types. We showed experimentally that 2-literal programs that are purely negative and constraint-free are harder than programs of any other type that our model can generate. We observed an easy-hard-easy pattern as we plotted the average number of choice points made by randomly generated programs that are purely negative and constraint free ($[mR^{-}]_n$) with increasing density. Random programs from the hard region that are purely negative and constraint free with 600 atoms are currently beyond the

reach of ASP solvers. Understanding the source of difficulty of these hard logic programs may help design better solvers.

We computed experimentally the probability of existence of an answer set for randomly generated programs with a fixed number of atoms and as a function of the density. We were able to approximate theoretically the probability of existence of an answer set for programs with very few rules as well as for dense programs with a large number of rules.

We further noticed that the purely negative constraint-free logic programs are tight logic programs, thus answer sets of these programs are exactly models of their completions. Formulas of the completion of a program from $[mR^-]_n$ are of the form (1) $a \vee b$, where $a \leftarrow not(b) \in P$, and (2) $\neg a \vee \neg b_1 \vee \ldots \vee \neg b_k$, where $a \leftarrow not(b_i)$, $1 \leq i \leq k$, are *all* rules in P with a as the head. Thus, the completions of such programs are simple formulas with most of their clauses consisting of two literals and all other clauses being disjunctions of atoms. These formulas are special mixed Horn formulas (MHFs).

We then defined models of these simple classes of MHFs with further restrictions on their syntactic structure. We would like to note that finding the right model is non-trivial (c.f. early model proposed in SAT [36]). We defined the following classes of MHFs: $CMH(k)$, $CMH^1(k)$, $MH^1(k)$ and $MH(k)$. The key finding is that despite their simple form, randomly generated formulas from these classes (for the appropriate selections of parameters) are challenging benchmarks for the current generation of state-of-the-art SAT solvers. Thus, formulas in these classes are relevant for the design of fast SAT solvers and deserve attention. We studied these classes experimentally, focusing on identifying phase transitions and hardness patterns, in order to facilitate generation of hard formulas. We observed a rapid phase transition for formulas generated from the class $CMH_n(k)$ with a fixed k and n, but with an increasing density of 2-literal rules, similar to the phase transition observed in random 3-SAT. We further showed the existence of an easy-hard-easy pattern as we plotted the average number of choice points generated at the critical region by the SAT solver *clasp* on programs from $CMH_N(k)$, with increasing values of k. We observed that the peak in the hard region occurred for programs generated in the phase transition region of $CMH_n(15)$ when the number of 2-clauses was about $14.5n$. We showed that

these programs pose a major challenge for the current generation of SAT solvers. Similarly, the instances from $CMH_n^1(k)$ show an easy-hard-easy behavior (as the length k of purely negative clauses grows), with the peak hardness when $k = 15$. The instances generated from $CMH_n^1(15)$ are predominantly satisfiable (probability of a random formula generated from that class being satisfiable is at least 0.9). We show that these satisfiable instances are very hard for local-search SAT solvers. We note that the class MH_n^1 is closely related to the class $[mR^-]_n$ of logic programs that we have studied, and identified as containing programs that are especially hard for the current generation of the answer set solvers.

We thus have generated hard instances for SAT and ASP solvers, and have analyzed the experimental and theoretical properties of these hard instances. Our research suggests several interesting directions for future investigations. Theoretical problems of interest include:

1. Studying the threshold value for MHFs generated from the model. Specifically, proving the existence of a threshold value and estimating its location.

2. Developing stronger conditions on the density over which programs in $[mR^-]_n$ a.a.s. have an answer set.

3. Providing a more precise explanation of the easy-hard-easy pattern that emerges for programs in $[mR^-]_n$ (and several other classes of programs and theories we considered)

Among problems of more practical importance are:

1. Developing local search solvers that can successfully terminate on theories from the class $CMH_n^1(k)$. Current solvers are well tuned for handling randomly generated satisfiable 3-CNF theories but fail on MHFs that we generate from the model $CMH_n^1(k)$.

2. Studying heuristics and conflict clause learning methods that would work well for programs and theories that can be generated from the hard regions for the models $[mR^-]_n$ and $CMH_n(k)$. Some of these techniques may prove also for programs and theories of other types.

Appendix A Experimental results on random logic programs from $[mR^-]_n$

We provide here the experimental results for programs from the class $[mR^-]_n$ using ASP solvers *clasp* and *smodels*. These experiments were performed using an AMD Athlon(tm) 64 X2 Dual Core Processor 5000+ with 512 KB of cache memory. The graphs in Figure A.1, Figure A.3, and Figure A.5, shows the easy-hard-easy pattern as we plot the average number of choice points generated by the solvers *clasp* and *smodels* on computing answer sets of satisfiable instances from $[mR^-]_n$ with increasing rule density $d = m/n$. Similarly, the graphs in Figure A.2, Figure A.4, and Figure A.6, show the easy-hard-easy pattern as we plot the average number of choice points generated by the solvers *clasp* and *smodels* for unsatisfiable instances from $[mR^-]_n$ with increasing rule density $d = m/n$.

In Figure A.7 and Figure A.8, we observe an easy-hard-easy pattern as we plot the average time taken by the solvers *smodels* and *clasp* for satisfiable and unsatisfiable instances from the class $[mR^-]_n$, with $n = 200$ and increasing rule density d.

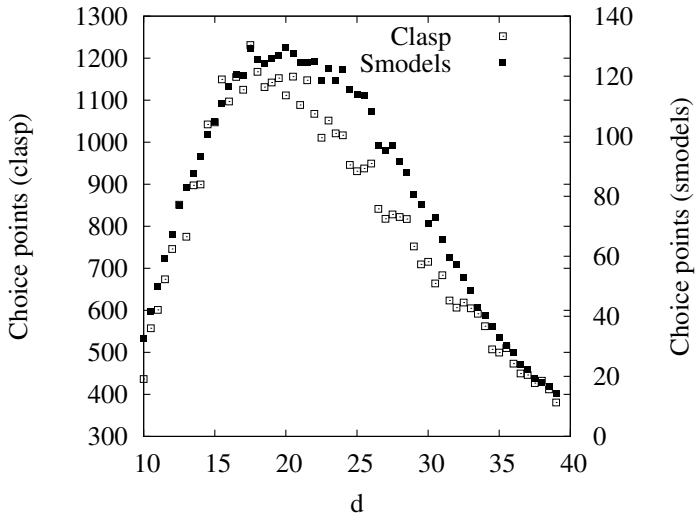


Figure A.1: The easy-hard-easy pattern for 500 consistent programs from the class $[mR^-]_n$ with $n = 125$.

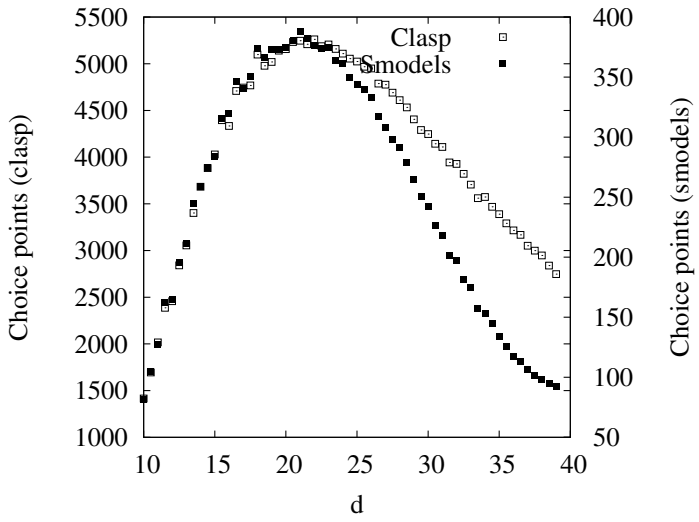


Figure A.2: The easy-hard-easy pattern for 100 inconsistent programs from the class $[mR^-]_n$ with $n = 125$.

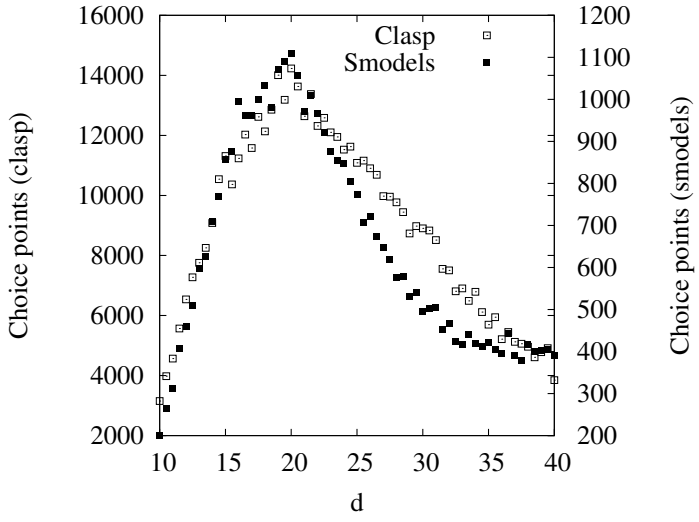


Figure A.3: The easy-hard-easy pattern for 500 consistent instances from the class $[mR^-]_n$ with $n = 175$.

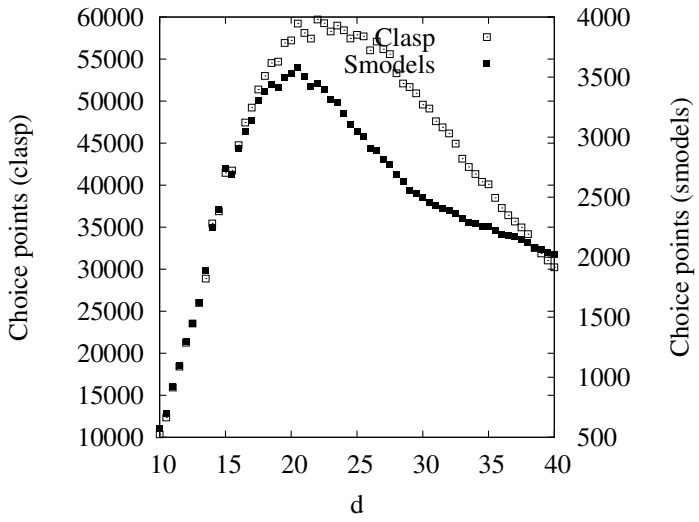


Figure A.4: The easy-hard-easy pattern for 100 inconsistent instances from the class $[mR^-]_n$ with $n = 175$.

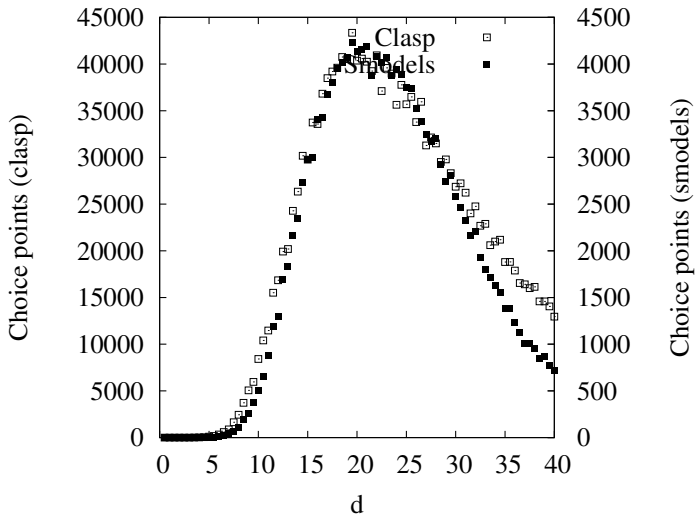


Figure A.5: The easy-hard-easy pattern for 500 consistent instances from the class $[mR^-]_n$ with $n = 200$.

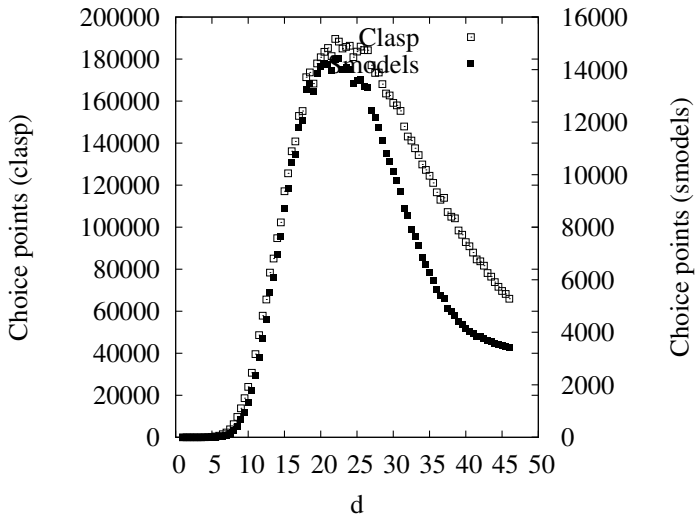


Figure A.6: The easy-hard-easy pattern for 100 inconsistent instances from the class $[mR^-]_n$ with $n = 200$.

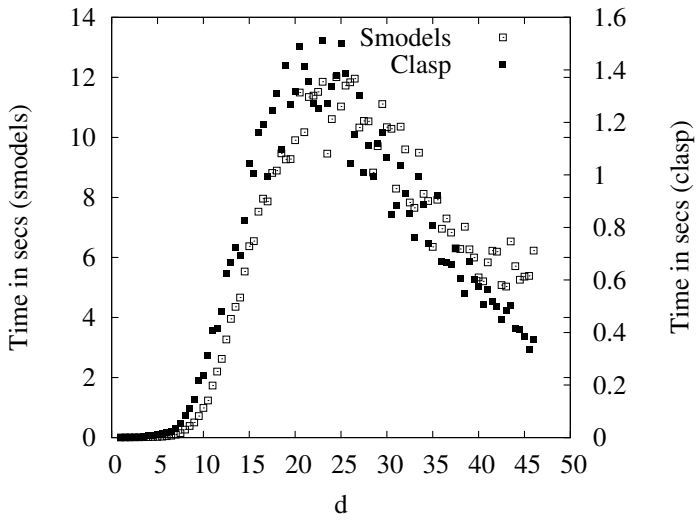


Figure A.7: The easy-hard-easy pattern for 100 consistent instances from the class $[mR^-]_n$ with $n = 200$.

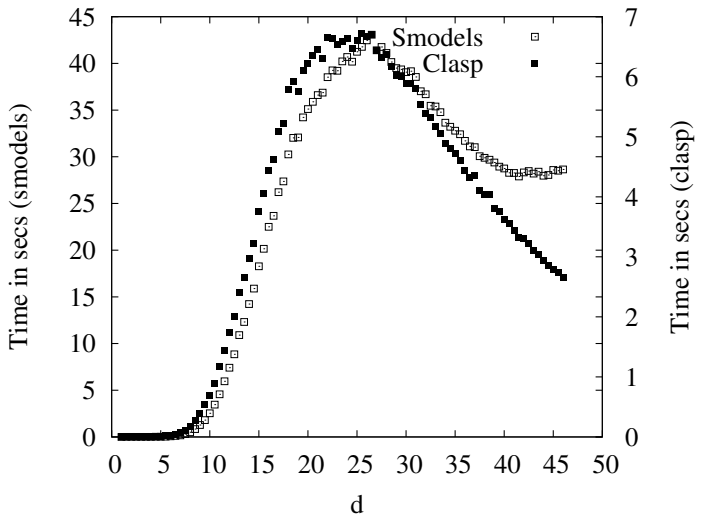


Figure A.8: The easy-hard-easy pattern for 100 inconsistent instances from the class $[mR^-]_n$ with $n = 200$.

Appendix B Experimental results on MHFs from $MH^n(k)$

We provide in Figures B.1-B.9, the experimental results obtained using the solver *clasp* on instances from the class $MH_n(k)$ with $k = 5, 10$, and $n = 50, 100, 150, 200, 250$. Figures B.10-B.13 gives the experimental results obtained using the solver *minisat* on instances from the class $MH_n(k)$ with $k = 10, 20, \ldots, 40$, and $n = 200$. These results show the existence of a phase-transition for the probability of existence of a model, and a corresponding easy-hard-easy pattern (as we plot the average number of choice points generated by the solver) for instances with increasing rule density d.

We also show the easy-hard-easy pattern obtained by plotting the average time taken by the solver *glucose* on instances from the class $MH^1_n(k)$ with $n = 200$ in Figure B.14, as well as by the solver *march_hi* on instances from the class $MH^1_n(k)$ with $n = 200$ in Figure B.15.

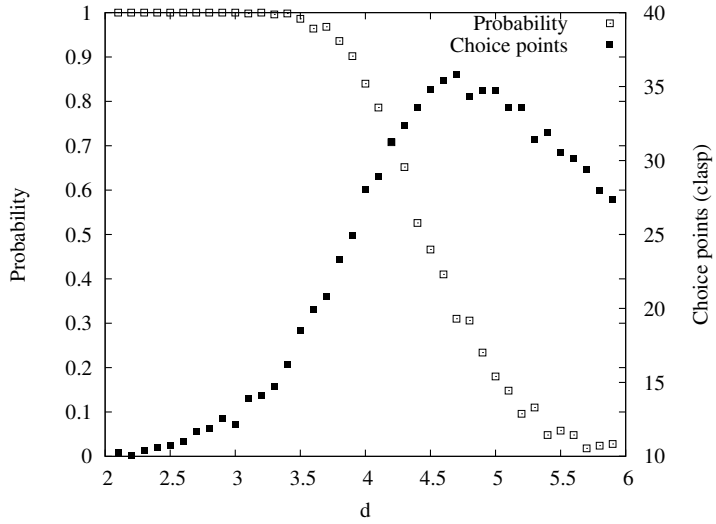


Figure B.1: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(5)$ with $n = 50$.

These results shown here were obtained using an AMD Athlon(tm) 64 X2 Dual Core Processor 5000+ with 512 KB of cache memory.

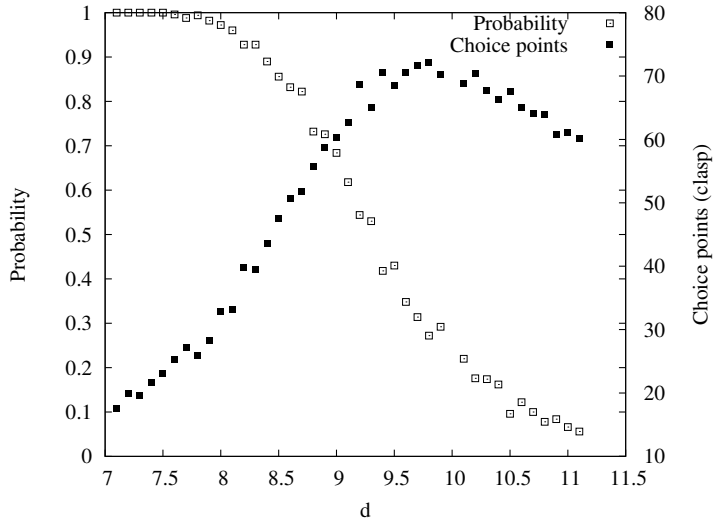


Figure B.2: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(10)$ with $n = 50$.

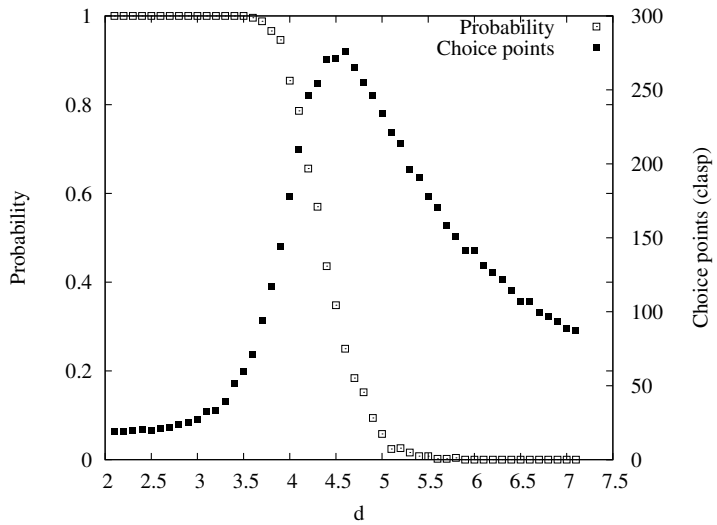


Figure B.3: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(5)$ with $n = 100$.

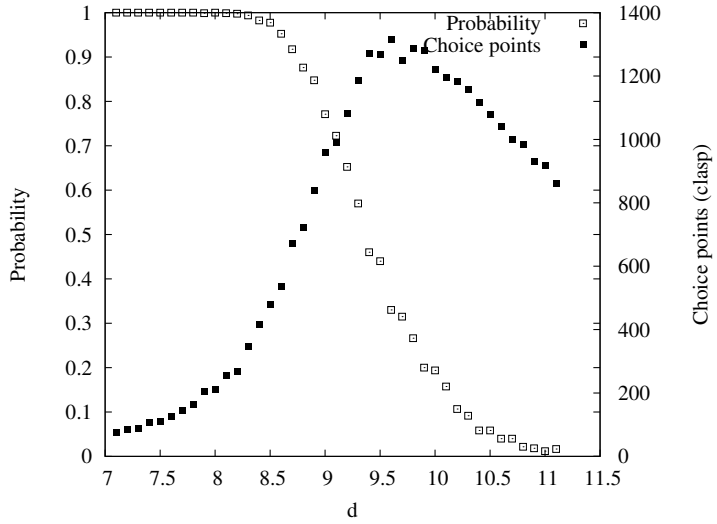


Figure B.4: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(10)$ with $n = 100$.

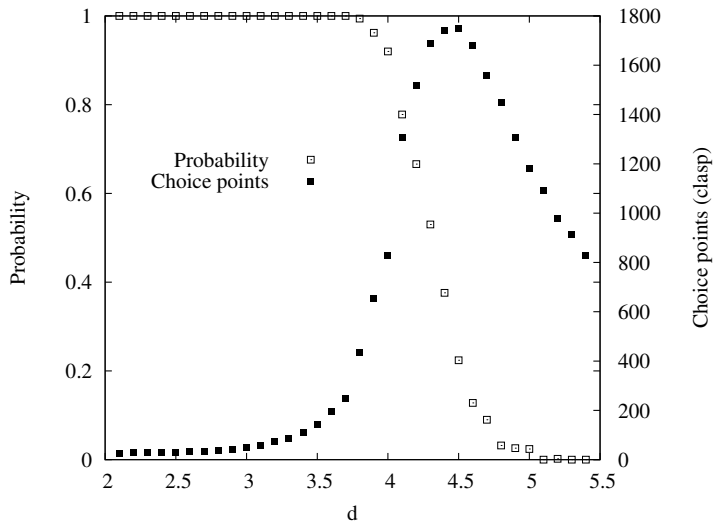


Figure B.5: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(5)$ with $n = 150$.

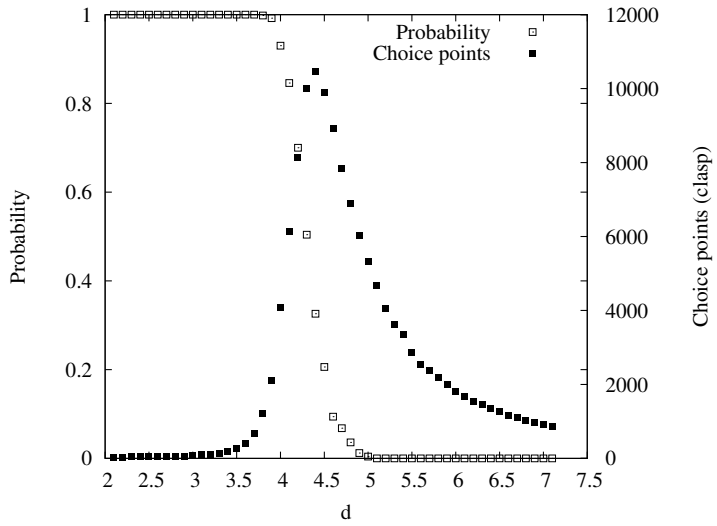


Figure B.6: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(5)$ with $n = 200$.

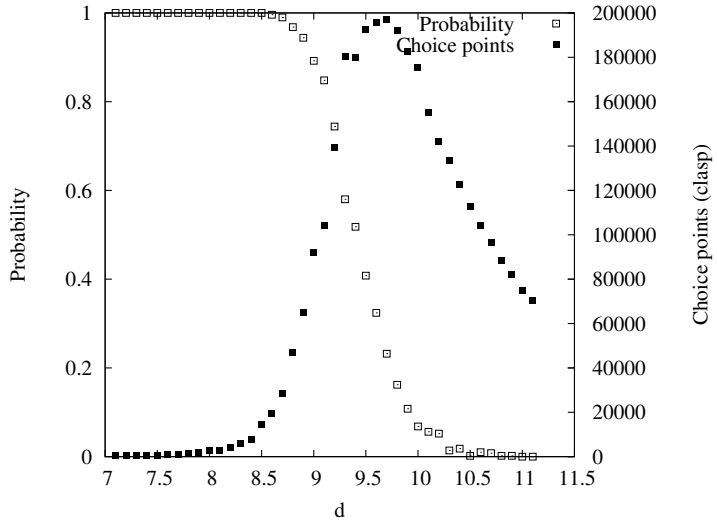


Figure B.7: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(10)$ with $n = 200$.

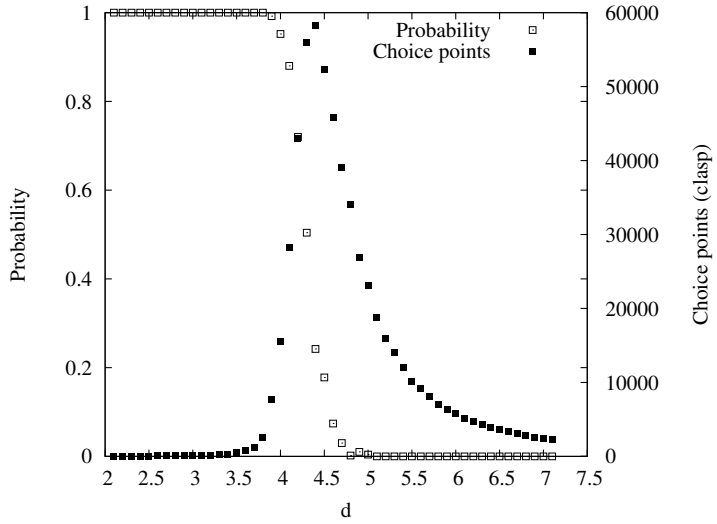


Figure B.8: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(5)$ with $n = 250$.

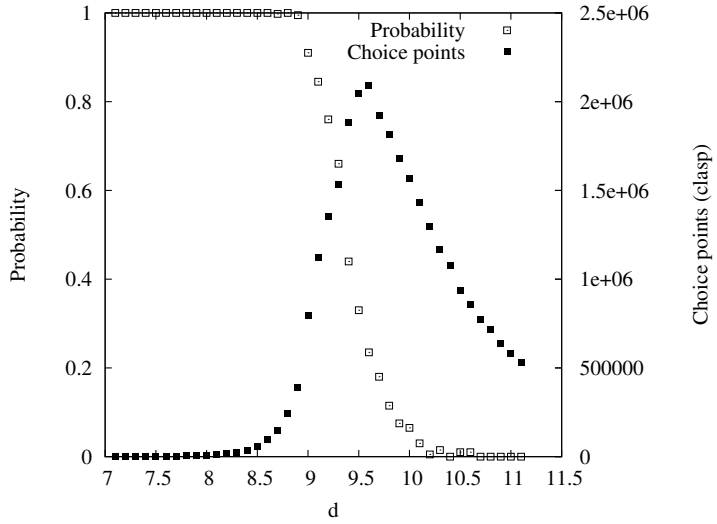


Figure B.9: The phase-transition and easy-hard-easy pattern for *clasp* on 500 instances from the class $MH_n(10)$ with $n = 250$.

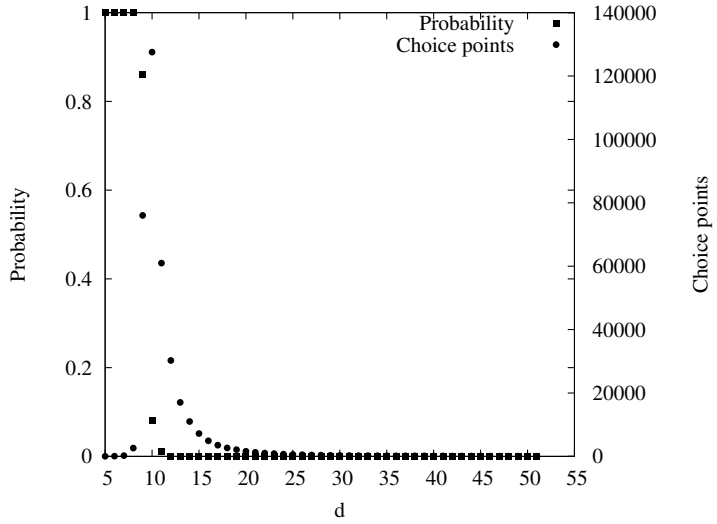


Figure B.10: The easy-hard-easy pattern for *Minisat* on 100 instances from the class $MH_n(10)$ with $n = 200$.

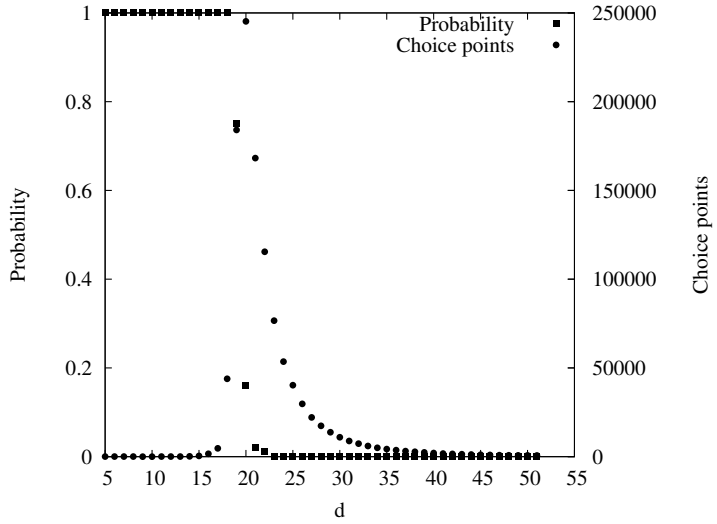


Figure B.11: The easy-hard-easy pattern for *Minisat* on 100 instances from the class $MH_n(20)$ with $n = 200$.

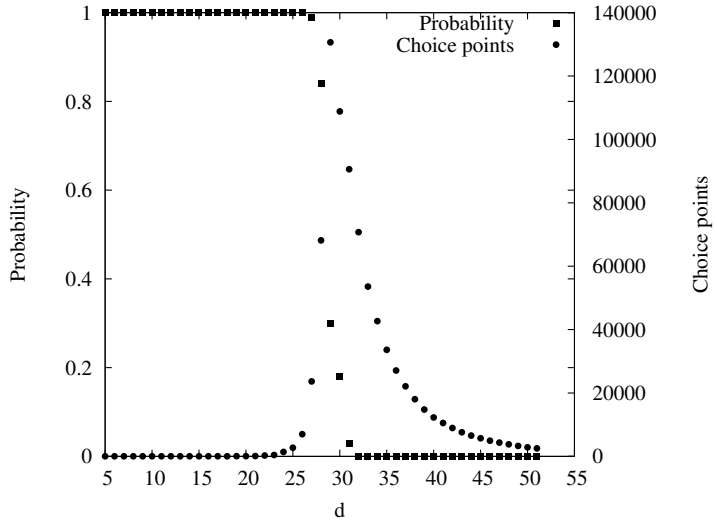


Figure B.12: The easy-hard-easy pattern for *Minisat* on 100 instances from the class $MH_n(30)$ with $n = 200$.

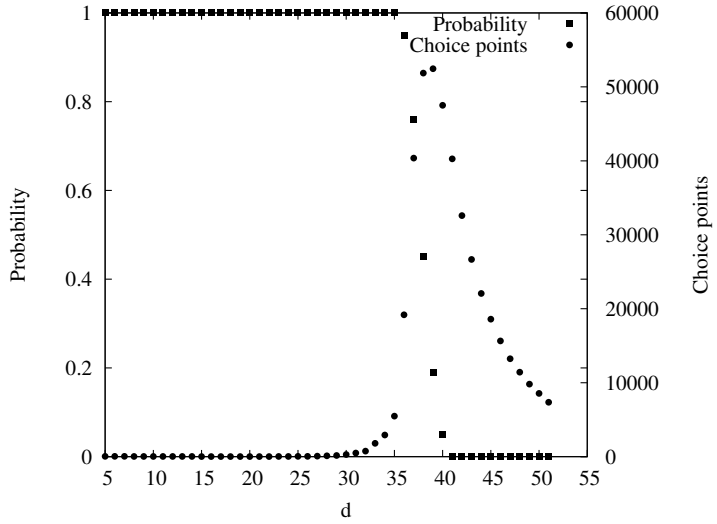


Figure B.13: The easy-hard-easy pattern for *Minisat* on 100 instances from the class $MH_n(40)$ with $n = 200$.

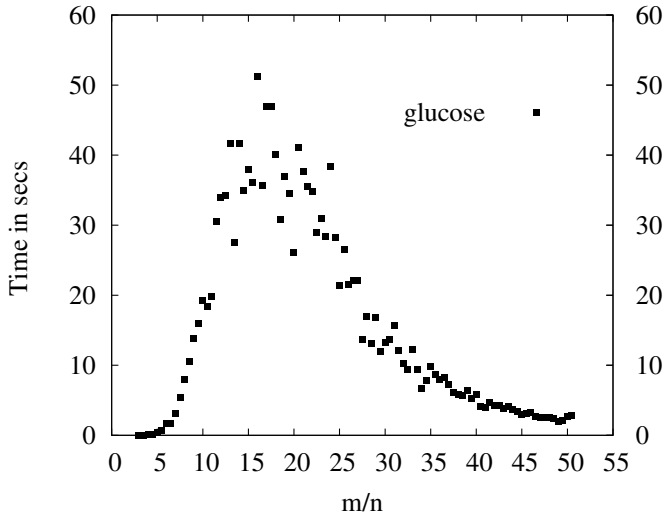


Figure B.14: The easy-hard-easy pattern for *glucose* on 100 instances from the class $MH_n^1(k)$ with $n = 200$.

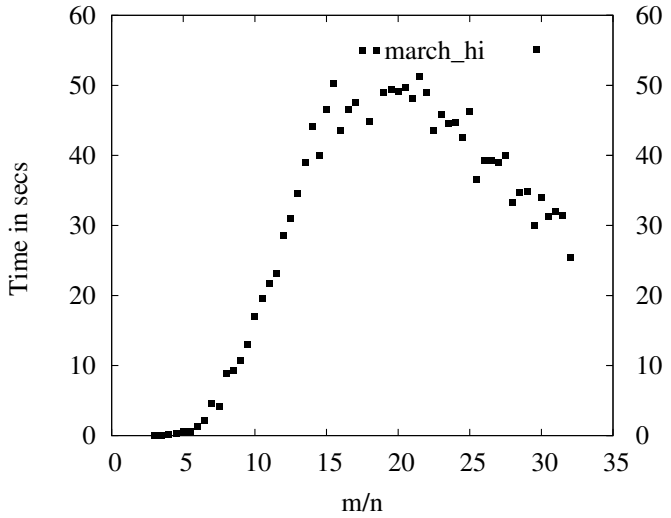


Figure B.15: The easy-hard-easy pattern for *march_hi* on 50 instances from the class $MH_n^1(k)$ with $n = 250$.

Bibliography

[1] *Asparagus*. http://asparagus.cs.uni-potsdam.de/.

[2] *The SAT competitions*. http://www.satcompetition.org/.

[3] *The First ASP System Competition*, 2007. http://asparagus.cs.uni-potsdam.de/contest/.

[4] *The SAT 2009 competition*, 2009. http://www.satcompetition.org/.

[5] *The Second ASP System Competition*, 2009. http://dtai.cs.kuleuven.be/events/ASP-competition/index.shtml.

[6] D. ACHLIOPTAS AND C. MOORE, *The asymptotic order of the random k-SAT threshold*, in FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science, Washington, DC, USA, 2002, IEEE Computer Society, pp. 779–788.

[7] D. ACHLIOPTAS AND C. MOORE, *Random k-SAT: Two moments suffice to cross a sharp threshold*, SIAM J. Comput., 36 (2006), pp. 740–762.

[8] D. ACHLIOPTAS AND Y. PERES, *The threshold for random k-SAT is 2k (ln 2 - O(k))*, in STOC '03: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, 2003, pp. 223–231.

[9] B. ASPVALL, M. F. PLASS, AND R. E. TARJAN, *A Linear-Time algorithm for testing the truth of certain quantified boolean formulas*, Inf. Process. Lett., 8 (1979), pp. 121–123.

[10] G. AUDEMARD AND L. SIMON, *Predicting learnt clauses quality in modern SAT solvers*, in IJCAI'09: Proceedings of the 21st International Joint Conference on Artificial Intelligence, San Francisco, CA, USA, 2009, Morgan Kaufmann Publishers Inc., pp. 399–404.

[11] R. J. BAYARDO, JR. AND R. C. SCHRAG, *Using CSP look-back techniques to solve real-world SAT instances*, in Proceedings of the 14th National Conference on Artificial Intelligence and 9th Conference on Innovative applications of Artificial Intelligence, AAAI'97/IAAI'97, AAAI Press, 1997, pp. 203–208.

[12] A. BIERE, M. HEULE, H. VAN MAAREN, AND T. WALSH, eds., *Handbook of Satisfiability*, vol. 185 of Frontiers in Artificial Intelligence and Applications, IOS Press, 2009.

[13] B. BOLLOBS, *Random Graphs*, Academic Press, London, 1985.

[14] G. BREWKA, *Logic programming with ordered disjunction*, in Proceedings of the 18th National Conference on Artificial Intelligence, Menlo Park, CA, USA, 2002, American Association for Artificial Intelligence, pp. 100–105.

[15] G. BREWKA, I. NIEMELÄ, AND M. TRUSZCZYŃSKI, *Answer set optimization*, in Proceedings of the 18th International Joint Conference on Artificial Intelligence, Morgan Kaufmann Publishers, August 2003, pp. 867–872.

[16] V. CHVÁTAL AND B. REED, *Mick gets some (the odds are on his side)*, in FOCS, 1992, pp. 620–627.

[17] E. CLARKE, A. BIERE, R. RAIMI, AND Y. ZHU, *Bounded model checking using satisfiability solving*, Form. Methods Syst. Des., 19 (2001), pp. 7–34.

[18] E. CLARKE, M. TALUPUR, H. VEITH, AND D. WANG, *SAT based predicate abstraction for hardware verification*, in In Sixth International Conference on Theory and Applications of Satisfiability Testing, 2003.

[19] S. A. COOK, *The complexity of theorem-proving procedures*, in STOC '71: Proceedings of the 3rd Annual ACM Symposium on Theory of computing, New York, NY, USA, 1971, ACM, pp. 151–158.

[20] J. M. CRAWFORD AND L. D. AUTON, *Experimental results on the crossover point in random 3-SAT*, Artif. Intell., 81 (1996), pp. 31–57.

[21] D. W. CURRIE, A. J. HU, AND S. RAJAN, *Automatic formal verification of DSP software*, in DAC '00: Proceedings of the 37th Annual Design Automation Conference, New York, NY, USA, 2000, ACM, pp. 130–135.

[22] M. DAVIS, G. LOGEMANN, AND D. LOVELAND, *A machine program for theorem-proving*, Commun. ACM, 5 (1962), pp. 394–397.

[23] O. DUBOIS, Y. BOUFKHAD, AND J. MANDLER, *Typical random 3-SAT formulae and the satisfiability threshold*, in SODA '00: Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete algorithms, Philadelphia, PA, USA, 2000, Society for Industrial and Applied Mathematics, pp. 126–127.

[24] N. EÉN AND N. SÖRENSSON, *An extensible SAT-solver*, in SAT, 2003, pp. 502–518.

[25] T. EITER, N. LEONE, C. MATEIS, G. PFEIFER, AND F. SCARCELLO, *A deductive system for non-monotonic reasoning*, in LPNMR, 1997, pp. 364–375.

[26] M. H. V. EMDEN AND R. A. KOWALSKI, *The semantics of predicate logic as a programming language*, Journal of the ACM, 23 (1976), pp. 569–574.

[27] E. ERDEM, *Theory and applications of answer set programming*, PhD thesis, 2002. Supervisor-Lifschitz, Vladimir.

[28] 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.

[29] J. FRANCO, J. M. PLOTKIN, AND J. W. ROSENTHAL, *Correction to probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem*, Discrete Appl. Math., 17 (1987), pp. 295–299.

[30] J. W. FREEMAN, *Improvements to propositional satisfiability search algorithms*, PhD thesis, Philadelphia, PA, USA, 1995.

[31] A. FRIEZE AND S. SUEN, *Analysis of two simple heuristics on a random instance of k-SAT*, J. Algorithms, 20 (1996), pp. 312–355.

[32] M. GEBSER, B. KAUFMANN, A. NEUMANN, AND T. SCHAUB, *clasp : A conflict-driven answer set solver*, in LPNMR, 2007, pp. 260–265.

[33] M. GEBSER, T. SCHAUB, AND S. THIELE, *Gringo: a new grounder for answer set programming*, in Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 266–271.

[34] M. GELFOND AND V. LIFSCHITZ, *The stable model semantics for logic programming*, in Proceedings of the Fifth International Conference on Logic Programming, R. A. Kowalski and K. Bowen, eds., Cambridge, Massachusetts, 1988, The MIT Press, pp. 1070–1080.

[35] E. GIUNCHIGLIA, Y. LIERLER, AND M. MARATEA, *SAT-Based answer set programming*, in Proceedings of Nineteenth National Conference on Artificial intelligence, Menlo Park, CA, USA, 2003, American Association for Artificial Intelligence.

[36] A. GOLDBERG, *On the complexity of the satisfiability problem*, in Fourth Workshop on Automated Deduction, 1979, pp. 1–6.

[37] C. P. GOMES, B. SELMAN, AND H. KAUTZ, *Boosting combinatorial search through randomization*, AAAI Press, 1998, pp. 431–437.

[38] M. HAJIAGHAYI AND G. B. SORKIN, *The satisfiability threshold for random 3-SAT is at least 3.52*, tech. report, IBM Research, 2003.

[39] M. HEULE AND H. VAN MAAREN, *march_hi*, 2009. http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf.

[40] S. HEYMANS, D. V. NIEUWENBORGH, AND D. VERMEIR, *Nonmonotonic ontological and rule-based reasoning with extended conceptual logic programs*, in ESWC, 2005, pp. 392–407.

[41] G.-S. HUANG, X. JIA, C.-J. LIAU, AND J.-H. YOU, *Two-literal logic programs and satisfiability representation of stable models: A comparison*, in AI '02: Proceedings

of the 15th Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence, London, UK, 2002, Springer-Verlag, pp. 119–131.

[42] S. JANSON, T. LUCZAK, AND A. RUCIŃSKI, *Random Graphs*, Wiley-Interscience, 2000.

[43] A. KAMATH, R.MOTWANI, K. PALEM, AND P. SPIRAKIS, *Tail bounds for occupancy and the satisfiability threshold conjecture*, Foundations of Computer Science, Annual IEEE Symposium on, 0 (1994), pp. 592–603.

[44] H. KAUTZ AND B. SELMAN, *Planning as satisfiability*, in ECAI '92: Proceedings of the 10th European Conference on Artificial Intelligence, New York, NY, USA, 1992, John Wiley & Sons, Inc., pp. 359–363.

[45] P. KILBY, J. K. SLANEY, S. THIÉBAUX, AND T. WALSH, *Backbones and backdoors in satisfiability*, in AAAI, 2005, pp. 1368–1373.

[46] S. KOTTLER, M. KAUFMANN, AND C. SINZ, *A new bound for an NP-Hard subclass of 3-SAT using backdoors*, in SAT, vol. 4996 of Lecture Notes in Computer Science, Springer, 2008, pp. 161–167.

[47] Y. LIERLER AND M. MARATEA, *Cmodels-2: SAT-based answer set solver enhanced to non-tight programs*, in LPNMR, 2004, pp. 346–350.

[48] V. LIFSCHITZ, *What is answer set programming?*, Menlo Park, CA, USA, 2008, American Association for Artificial Intelligence, pp. 1594–1597.

[49] F. LIN AND Y. ZHAO, *ASSAT: computing answer sets of a logic program by SAT solvers*, in Eighteenth National Conference on Artificial intelligence, Menlo Park, CA, USA, 2002, American Association for Artificial Intelligence, pp. 112–117.

[50] L. LIU AND M. TRUSZCZYŃSKI, *Pbmodels - software to compute stable models by pseudoboolean solvers*, in LPNMR, 2005, pp. 410–415.

[51] L. LIU AND M. TRUSZCZYNSKI, *Properties of programs with monotone and convex constraints*, in Proceedings of the 20th National Conference on Artificial Intelligence, vol. 2, AAAI Press, 2005, pp. 701–706.

[52] I. LYNCE AND J. MARQUES-SILVA, *Hidden structure in unsatisfiable random 3-SAT: an empirical study*, in ICTAI '04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA, 2004, IEEE Computer Society, pp. 246–251.

[53] V. W. MAREK AND M. TRUSZCZYNSKI, *Stable models and an alternative logic programming paradigm*, CoRR, cs.LO/9809032 (1998).

[54] W. MAREK AND V. S. SUBRAHMANIAN, *The relationship between stable, supported, default and autoepistemic semantics for general logic programs*, Theor. Comput. Sci., 103 (1992), pp. 365–386.

[55] W. MAREK AND M. TRUSZCZYŃSKI, *Autoepistemic logic*, J. ACM, 38 (1991), pp. 587–618.

[56] M. MINOUX, *LTUR: a simplified linear-time unit resolution algorithm for horn formulae and computer implementation*, Inf. Process. Lett., 29 (1988), pp. 1–12.

[57] D. G. MITCHELL, B. SELMAN, AND H. J. LEVESQUE, *Hard and easy distributions for SAT problems*, in Proceedings of the Tenth National Conference on Artificial Intelligence, P. Rosenbloom and P. Szolovits, eds., Menlo Park, California, 1992, AAAI Press, pp. 459–465.

[58] M. W. MOSKEWICZ, C. F. MADIGAN, Y. ZHAO, L. ZHANG, AND S. MALIK, *Chaff: engineering an efficient SAT solver*, in Proceedings of the 38th Annual Design Automation Conference, DAC '01, New York, NY, USA, 2001, ACM, pp. 530–535.

[59] G. NAMASIVAYAM AND M. TRUSZCZYŃSKI, *Simple random logic programs*, in Proceedings of the 10th International Conference on Logic Programming and Nonmono-

tonic Reasoning, LPNMR '09, Berlin, Heidelberg, 2009, Springer-Verlag, pp. 223–235.

[60] G. NAMASIVAYAM AND M. TRUSZCZYŃSKI, *Simple but hard mixed horn formulas*, in SAT, 2010, pp. 382–387.

[61] 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, eds., 1998, pp. 72–79.

[62] I. NIEMELÄ AND P. SIMONS, *Smodels – an implementation of the stable model and well-founded semantics for normal logic programs*, in Proceedings of LPNMR, Springer-Verlag, 1997, pp. 420–429.

[63] S. PORSCHEN, T. SCHMIDT, AND E. SPECKENMEYER, *On some aspects of mixed horn formulas*, in SAT '09: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, Berlin, Heidelberg, 2009, Springer-Verlag, pp. 86–100.

[64] S. PORSCHEN AND E. SPECKENMEYER, *Worst case bounds for some NP-Complete modified Horn-SAT problems*, in SAT (Selected Papers), vol. 3542 of Lecture Notes in Computer Science, Springer, 2005, pp. 251–262.

[65] R. REITER, *A logic for default reasoning*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987, pp. 68–93.

[66] T. J. SCHAEFER, *The complexity of satisfiability problems*, in Proceedings of the 10th Annual ACM Symposium on Theory of Computing, STOC '78, New York, NY, USA, 1978, ACM, pp. 216–226.

[67] A. L. SELMAN, *A taxonomy of complexity classes of functions*, Journal of Computer and System Sciences, 48 (1992), pp. 357–381.

[68] B. SELMAN, H. KAUTZ, AND B. COHEN, *Local search strategies for satisfiability testing*, in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1995, pp. 521–532.

[69] B. SELMAN, H. LEVESQUE, AND D. MITCHELL, *A new method for solving hard satisfiability problems*, in AAAI, 1992, pp. 440–446.

[70] P. SIMONS, *Extending the stable model semantics with more expressive rules*, in Logic Programming and Non-monotonic Reasoning, 1999, pp. 305–316.

[71] T. SOININEN AND I. NIEMELÄ, *Developing a declarative rule language for applications in product configuration*, in PADL '99: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages, London, UK, 1998, Springer-Verlag, pp. 305–319.

[72] T. SOININEN AND I. NIEMELÄ, *Developing a declarative rule language for applications in product configuration*, in PADL'99, vol. 1551 of Lecture Notes in Computer Science, 1999, pp. 305–319.

[73] R. M. STALLMAN AND G. J. SUSSMAN, *Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis*, Artif. Intell., 9 (1977), pp. 135–196.

[74] T. SYRJÄNEN, lparse*, a procedure for grounding domain restricted logic programs*. http://www.tcs.hut.fi/Software/smodels/lparse/, 1999.

[75] T. SYRJÄNEN, *Including diagnostic information in configuration models*, in CL '00: Proceedings of the First International Conference on Computational Logic, London, UK, 2000, Springer-Verlag, pp. 837–851.

[76] M. TE CHAO AND J. FRANCO, *Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k-SAT problem*, Information Science, 51 (1990), pp. 289–314.

[77] L. XU, F. HUTTER, H. H. HOOS, AND K. LEYTON-BROWN, *SATzilla: portfolio-based algorithm selection for SAT*, J. Artif. Int. Res., 32 (2008), pp. 565–606.

[78] H. ZENG AND S. MCILRAITH, *Experimental results on the satisfiable core in random 3SAT*, in Ninth International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, USA, 2006.

[79] W. ZHANG, *Phase transitions and backbones of 3-SAT and maximum 3-SAT*, in Principles and Practice of Constraint Programming, 2001, pp. 153–167.

[80] Y. ZHAO AND F. LIN, *Answer set programming phase transition: A study on randomly generated programs*, in ICLP, 2003, pp. 239–253.

Vita

Gayathri Namasivayam

Date of birth April 1st, 1981

Place of birth Chennai, India

EDUCATION

- Master of Science, Computer Science
  University of Kentucky, Lexington, KY, May 2010

- Bachelor of Engineering, Computer Science
  University of Madras, Chennai, India, May 2002

TEACHING EXPERIENCE

- Instructor Summer 2008, 2009
  Department of Computer Science, University of Kentucky
  - Introduction to computers: Designed syllabus, taught basics of computers (hardware, software, operating systems, introduction to databases, introduction to networks, viruses, etc.), taught laboratory classes on Microsoft Office tools (Word, Excel, Access, Powerpoint), prepared both in-class exams and lab exams for students, and graded their work.

- Teaching Assistant Fall 2006 – Fall 2009
  Department of Computer Science, University of Kentucky
  Taught the laboratory classes for each of these courses and graded all lab assignments and exams.

- Introduction to computer programming: Topics ranged from introductory concepts (data types, functions, headers, pointers, classes) to programming using C++.
- Introduction to engineering computing: C++ programming oriented towards solving practical engineering problems; topics included all introductory concepts (data types, functions, headers, pointers, classes), and concepts of GUI programming.
- Introduction to program design, abstraction, and problem solving: Introduces the concepts of object-oriented programming; topics covered in the course included data structures, dynamic data and pointers, and recursion. In addition, the course introduces sorting and searching and adresses the complexity of algorithms.
- Discrete mathematics: Introduces the fundamental principles of computer science which include set theory, induction, functions, Boolean algebra, permutations, combinations, recurrences, and introductory graph theory.

RESEARCH EXPERIENCE

Research Assistant — Summer 2003 – Spring 2010

Department of Computer Science,

University of Kentucky, KY.

- Pseudo-Boolean solvers: Developing solvers that solve a system of linear inequalities by translating a system of linear equations into a logic program and using existing logic programming solvers to solve the equations.
- Knowledge representation and preference handling: Building a model of the welfare to work domain, specifying constraints and preferences in the welfare to work domain using a logical language, and developing logic based language for handling user preferences.

- Lookahead techniques in answer set programming (ASP) solvers: Improving existing algorithms and design of new algorithms for lookahead in ASP solvers.
- Random logic programs: Studying methods to generate random logic programs that are hard to solve by existing state-of-the-art logic programming solvers, and understanding the properties of these hard random programs.
- Random SAT: Studying methods to generate hard random mixed Horn formulas for both complete and incomplete solvers, and analyzing the properties of these hard formulas.

PUBLICATIONS

- Gayathri Namasivayam, Miroslaw Truszczynski: *A Smodels System with Limited Lookahead Computation.* LPNMR 2007: 278-283.
- Martin Gebser, Lengning Liu, Gayathri Namasivayam, Andre Neumann, Torsten Schaub, Miroslaw Truszczynski: *The First Answer Set Programming System Competition.* LPNMR 2007: 3-17.
- Gayathri Namasivayam: *PB-smodels a Pseudo-Boolean Solver.* AAAI 2006.
- Gayathri Namasivayam: *Study of Random Logic Programs.* ICLP 2009: 555-556.
- Gayathri Namasivayam, Miroslaw Truszczynski: *Simple Random Logic Programs.* LPNMR 2009: 223-235.
- Gayathri Namasivayam, Miroslaw Truszczynski: *Simple but Hard Mixed Horn Formulas.* SAT 2010: 382-387.

COMPETITIONS

Submitted benchmark problems and solvers to pseudo-boolean evaluations held as a part of SAT 2006 and SAT 2007, and to the second answer set programming competition held as a part of LPNMR 2009.

AWARDS

- Federal Logic Conference scholarship in 2010.
- International Conference in Logic Programming - Doctoral Consortium (ICLP-DC) Scholarship in 2009.
- Google scholarship for women engineers in 2008.
- Kentucky Graduate Scholarship from Fall 2002 to present.
- Logic Programming and Non-monotonic Reasoning (LPNMR) scholarship, in 2007.
- American Association of Artificial Intelligence (AAAI 2006) scholarship, in 2006.
- Ranked 2nd in the Computer Science department, R.M.K. Engineering college, for the academic year 2000-2001.

PROFESSIONAL ACTIVITIES

- Organized as a team member the first ASP competition, which was held as a part of the International Conference on Logic Programming and Non-monotonic Reasoning 2007; drafted the competition rules, and built software that can verify the correctness of the solutions provided by the competing ASP solvers on problems modeled as logic programs.
- Participated at the engineering day celebrations in 2006 and 2008, held at the University of Kentucky; presented to high school students the use of artificial intelligence techniques to build and solve sudoku puzzles; motivated them to pursue a career in computer science engineering.
- Organized a national conference on information technology BROUTER 2001, that was held at the R.M.K. Engineering College, University of Madras; organized different events such as paper presentations and quizes; developed and maintained a web site that allows members to register online and provides information about the conference, and oversaw the entire event.

- Presented papers and posters at AAAI 2006, LPNMR 2007, the International Conference on Artificial Intelligence (ICAI) 2007, ICLP 2009, and SAT 2010.