

UNIVERSITY OF MINNESOTA

This is to certify that I have examined this copy of a master's thesis by

**ASHUTOSH NAGLE**

and have found that it is complete and satisfactory in all respects,  
and that any and all revisions required by the final  
examining committee have been made.

**Dr. Hudson Turner**

---

Name of Faculty Advisor

---

Signature of Faculty Advisor

---

Date

GRADUATE SCHOOL

**A Finite Domain Satisfiability Solver with Negation**

A THESIS  
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL  
OF THE UNIVERSITY OF MINNESOTA  
BY

Ashutosh Nagle

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
MASTER OF SCIENCE

August 2004

## **Acknowledgements**

I would like to thank Dr. Turner for the invaluable guidance he offered throughout this thesis. The opportunities he made available to me taught me a lot. He was always there to help me whenever I needed it. He is the one who saw me through this challenging yet interesting journey. I am grateful to him for making my thesis such a great success. This thesis has been supported in part by NSF grant #0091773.

I would also like to thank my committee members, Dr. Dunham and Dr. Froncek, for their time and suggestions.

Thanks to the staff and the faculty of the Computer Science Department at UMD for their support and encouragement during the past two years.

Dedicated to my parents  
Mrs. Rashmi Nagle and Mr. Arvind Nagle  
and  
my fiancée  
Kalyani.

## Abstract

Boolean satisfiability (SAT) is the archetypal NP-complete problem and is useful for solving NP-complete problems in planning, digital circuits, software debugging, and other application areas. A Boolean SAT problem consists of a set of “clauses” – expressions that place restrictions on the valuation (“interpretation”) of a set of Boolean variables. To solve a Boolean SAT problem, one must either find an interpretation that satisfies the restrictions or determine that no such interpretation exists. The underlying algorithm in most Boolean SAT solvers is Davis-Putnam-Logemann-Loveland (DPLL), which is essentially a modified depth-first heuristic search for a satisfying interpretation.

Sinha extended the DPLL algorithm to handle variables whose range of values is not restricted to the Boolean case, but instead encompasses some finite set like  $\{1, 2, 3\}$  or  $\{red, yellow, green\}$ . Although such problems can be solved by reduction to Boolean SAT, Sinha demonstrated that it may be computationally advantageous to solve them directly. In this thesis we extend Sinha’s finite domain DPLL algorithm to handle a still richer language, and demonstrate the computational advantage of this extension. We also compare the performance of five different search heuristics in our implementation of the finite domain DPLL algorithm.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Boolean Satisfiability . . . . .	1
1.2	Finite Domain Satisfiability . . . . .	2
1.3	Contribution of this Thesis . . . . .	3
1.4	Other Related Work . . . . .	3
1.5	Outline of the Thesis . . . . .	5
<b>2</b>	<b>Translations of Finite Domain Theories</b>	<b>6</b>
2.1	Eliminating Negation . . . . .	6
2.2	Boolean Translations . . . . .	7
2.2.1	Quadratic Translation . . . . .	7
2.2.2	Linear Translation . . . . .	8
2.2.3	Remarks on Boolean Translations . . . . .	11
<b>3</b>	<b>Finite Domain DPLL</b>	<b>13</b>
3.1	Partial Interpretation . . . . .	13
3.2	Depth-first Search . . . . .	14
3.3	Reduction of Theory . . . . .	16
3.4	Safe and Obvious Assignments . . . . .	17
3.4.1	Unit Clause Propagation (UCP) . . . . .	19
3.4.2	Entailment Handling . . . . .	20
3.4.3	Pure Literal Handling . . . . .	20
3.5	Finite Domain DPLL – Algorithm . . . . .	22
3.6	Heuristics . . . . .	22
3.6.1	Heuristic 1: Highest Satisfaction Under Maximum Impact . . . . .	22

3.6.2	Heuristic 2: Highest (Satisfaction - Falsification)	24
3.6.3	Heuristic 3: Highest (Satisfaction - Falsification) <i>Negative Literals Only</i>	25
3.6.4	Heuristic 4: Highest (Satisfaction - Falsification) <i>Atoms Only</i>	25
3.6.5	Heuristic 5: Highest Satisfaction	25
3.7	Examples	26
3.7.1	Example 1:	26
3.7.2	Example 2:	28
<b>4</b>	<b>Implementation</b>	<b>32</b>
4.1	Data Structures	32
4.1.1	Literal	33
4.1.2	Clause	33
4.1.3	Literal Pool Node (LPN)	34
4.1.4	Literal Pool (LP)	34
4.1.5	Set	35
4.1.6	Class Diagram	35
4.2	Example	35
4.3	Functionalities	38
4.3.1	Empty theory detection	38
4.3.2	Empty clause detection	38
4.3.3	Unit clause detection	38
4.3.4	Entailment handling	39
4.3.5	Pure literal detection	39
4.3.6	Branching: Computation of heuristic	40
4.3.7	Backtracking: Avoiding multiple copies of the theory	41

<b>5</b>	<b>Evaluation Procedure and Experimental Results</b>	<b>43</b>
5.1	Syntax . . . . .	43
5.2	Evaluation Procedure . . . . .	45
5.2.1	Heuristics Comparison . . . . .	45
5.2.2	Comparison with and without Negative Literals Compiled Out . . . . .	54
5.2.3	Comparison with Sinha’s Solver . . . . .	59
5.2.4	Comparison with zChaff Solver . . . . .	62
<b>6</b>	<b>Conclusion</b>	<b>65</b>
6.1	Future Work . . . . .	65

## List of Figures

1	Pseudo code of DF Algorithm . . . . .	15
2	Search Tree . . . . .	15
3	Final Search Tree . . . . .	16
4	Pseudo code of DFR Algorithm . . . . .	18
5	pseudo code of DPLL Algorithm . . . . .	23
6	Example 1: Number of Clauses for Each Literal . . . . .	26
7	Search Tree - Heuristic 1 on Example 1 . . . . .	27
8	Search Tree - Heuristic 5 on Example 1 . . . . .	28
9	Example 2: Number of Clauses for Each Literal . . . . .	29
10	Search Tree - Heuristic 1 on Example 2 . . . . .	30
11	Search Tree - Heuristic 5 on Example 2 . . . . .	31
12	Literal . . . . .	33
13	Clause . . . . .	33
14	Literal Pool Node (LPN) Class . . . . .	34
15	Literal Pool (LP) Class . . . . .	35
16	Set Class . . . . .	35
17	Class Diagram . . . . .	36
18	Example of Data Structures . . . . .	37
19	Sample CNF file . . . . .	44
20	Heuristics Comparison for $r = 5$ . . . . .	47
21	Heuristics Comparison for $r = 10$ . . . . .	49
22	Heuristics Comparison for $r = 10$ . . . . .	50
23	Comparison of Heuristic Family ( $\alpha \times \text{satisfaction} \pm \text{falsification}$ ) for $r = 10$ . . . . .	52
24	Comparison of Heuristic Family ( $\alpha \times \text{satisfaction} \pm \text{falsification}$ ) for $r = 10$ . . . . .	53

25	Comparison with and without Negative Literals with Heuristic 4 and $r = 5$ . . . . .	55
26	Comparison with and without Negative Literals with Heuristic 4 and $r = 10$ . . . . .	56
27	Comparison with and without Negative Literals with Heuristic 5 and $r = 5$ . . . . .	57
28	Comparison with and without Negative Literals with Heuristic 5 and $r = 10$ . . . . .	58
29	Comparison with Sinha's Solver with $r=5$ and Heuristic 5 . . . . .	60
30	Comparison with Sinha's Solver with $r=10$ and Heuristic 5 . . . . .	61
31	Comparison with zChaff Solver with $r=5$ and Heuristic 4 . . . . .	63
32	Comparison with zChaff Solver with $r=10$ and Heuristic 4 . . . . .	64

## List of Tables

1	Search Time in Seconds for $r = 5$ . . . . .	46
2	Number of Backtracks for $r = 10$ . . . . .	48
3	Search Time in Seconds for $r = 10$ . . . . .	49
4	Number of Backtracks for $r = 10$ . . . . .	51
5	Search Time in Seconds for $r = 10$ . . . . .	51
6	Search Time in Seconds for Heuristic 4 . . . . .	54
7	Search Time in Seconds for Heuristic 5 . . . . .	55
8	Execution Time in Seconds with Heuristic 5 . . . . .	59
9	Execution Time in Seconds with Heuristic 4 . . . . .	62

# 1 Introduction

Satisfiability (SAT) is the prototypical NP-complete problem [24]. Many NP-complete problems arising in different fields – such as artificial intelligence, graph theory, logic circuit design and testing, cryptography, database systems, hardware and software verification, planning, scheduling and model checking – can be solved relatively effectively by reduction to SAT [23, 26, 22, 21]. SAT solvers [13, 16, 14] have seen a great improvement in recent years, allowing larger problem instances in different application domains to be solved. (One can find more about current SAT solvers and satisfiability benchmark problem instances at <http://www.satlive.org/>.) This thesis takes as its starting point work by Sinha [25] on extending the SAT problem to allow for variables that can take values from arbitrary finite domains, in addition to the standard Boolean variables.

## 1.1 Boolean Satisfiability

Let's begin by defining Boolean satisfiability. A *signature* is a finite set of symbols, called *atoms*. A *literal* is an atom or its *negation*, i.e. the atom preceded by  $\neg$ . A *clause* is a finite set of literals. A *theory* is a finite set of clauses. An *interpretation* is a function from atoms to truth values  $\{\mathbf{t}, \mathbf{f}\}$ . An interpretation  $I$  *satisfies* an atom  $p$  if and only if  $I(p) = \mathbf{t}$  and it satisfies  $\neg p$  if and only if  $I(p) = \mathbf{f}$ . An interpretation *satisfies* a clause if it satisfies at least one of the literals in the clause. An interpretation *satisfies* a theory if it satisfies all the clauses in it.

Solving a Boolean SAT problem involves determining whether or not a given theory is satisfiable, i.e. has a satisfying interpretation, or *model*. In many cases, in addition to this yes/no answer it's also required to find the model itself.

For example, consider the theory  $\{\{p, \neg q\}, \{q, \neg r\}, \{r, \neg p\}\}$  (over signature  $\{p, q, r\}$ ). The interpretation that maps each of  $p$ ,  $q$  and  $r$  to  $\mathbf{t}$  is one of the models of the theory. Another model maps  $p$ ,  $q$  and  $r$  all to  $\mathbf{f}$ . In fact these are the only models of this theory.

## 1.2 Finite Domain Satisfiability

A natural extension of Boolean satisfiability is finite domain satisfiability. Here the *signature* is a finite set of symbols, called *constants*, along with an associated finite set  $dom(c)$  for each constant  $c$ , called the *domain* of  $c$ . An *atom* is an expression of the form  $c=v$ , where  $c$  is a constant and  $v \in dom(c)$ . A *literal* is an atom  $c=v$  or its negation, written  $c \neq v$ . A *clause* is a finite set of literals. A *theory* is a finite set of clauses. An *interpretation* maps each constant to an element of its domain. An interpretation  $I$  *satisfies* an atom  $c=v$  if and only if  $I(c) = v$ , and it *satisfies* a literal  $c \neq v$  if and only if  $I(c) \neq v$ . An interpretation *satisfies* a clause if it satisfies at least one of its literals, and *satisfies* a theory if it satisfies all the clauses in the theory. A theory is *satisfiable* if there is at least one interpretation that satisfies it.

As with Boolean satisfiability, solving a finite domain SAT problem involves determining whether or not given theory is satisfiable, and if it is, then finding a satisfying interpretation or *model* of the theory.

For example, consider the theory consisting of the following clauses, where the signature is  $\{A, B, C\}$ , with domains  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ .

$$\begin{aligned} &\{A \neq 0, B = 2\} \\ &\{A = 0, B = 1\} \\ &\{A = 1, B \neq 2, C = 2\} \\ &\{A = 2, B \neq 1\} \\ &\{A = 1, B \neq 1\} \end{aligned} \tag{1}$$

This theory is satisfiable; the interpretation that maps  $A$  to 0,  $B$  to 2 and  $C$  to 2 satisfies the theory. In fact, this is the only model of the theory.

Boolean SAT can be seen as the special case of finite domain SAT in which all the atoms have the domain  $\{\mathbf{t}, \mathbf{f}\}$ , if we agree to write  $c$  as shorthand for  $c = \mathbf{t}$  and  $\neg c$  as shorthand for  $c = \mathbf{f}$ .

Finite domain satisfiability is particularly interesting because many problems are expressed and understood more naturally when using variables with finite domains. A finite domain SAT problem can be con-

verted into a Boolean SAT problem and solved using standard Boolean SAT solvers, however, in doing this, the problem instance gets bigger and its essential structure may be obscured.

### 1.3 Contribution of this Thesis

In this thesis we extend Sinha’s work [25]. Sinha implemented a finite domain SAT solver by generalizing a standard Boolean satisfiability algorithm, Davis-Putnam-Logemann-Loveland (DPLL) [18]. The DPLL algorithm is a form of heuristic search (for a model) and is the underlying algorithm of most Boolean SAT solvers. Sinha’s generalization of DPLL solves a finite domain SAT problem directly, without converting it to Boolean SAT. Sinha demonstrated that her finite domain SAT solver can outperform (in terms of execution time) one of the industry standard Boolean solvers – SATO [14].

Sinha’s solver is restricted to finite domain theories in which clauses contain only atoms: it does not handle negative literals, i.e. literals of the form  $c \neq v$ . One of the contributions of the current thesis is to extend Sinha’s finite domain DPLL algorithm to allow negative literals, and to implement this extension of the algorithm. If for some constant  $A$ ,  $dom(A) = \{1,2,3\}$ , then saying  $A \neq 2$  is the same as saying “ $A=1$  OR  $A=3$ ”. Here for one literal we require two atoms. Similarly if  $A$  has 100 elements in its domain, we will need 99 atoms to eliminate a negative literal. Experimental results demonstrate that this extension can offer a payoff in computation time, although it slightly complicates the algorithm and its implementation.

Since DPLL algorithms, including our finite domain extension of DPLL, involve heuristic search through the space of (partial) interpretations, we also experiment in this thesis with the heuristic used by the solver. We tried five alternative heuristics and observed that two of them performed better than the others.

### 1.4 Other Related Work

There has been quite a lot of research on SAT in recent years. Researchers in the field have tried various techniques to improve the performance of SAT solvers. Extensive research has been done on improving the data structures and heuristics used in the search. Zhang [17] gives a nice introductory account of recent work

of this kind. Another area where attempts are being made for performance improvement is the expressiveness of the input language. The work in this thesis is an example of this kind of work. Another example is work by Thiffault et al. [3] which involves an implementation of the DPLL algorithm for general Boolean formulas. (In general, Boolean formulas are built up recursively from Boolean atoms using standard propositional connectives:  $\neg, \wedge, \vee, \supset, \equiv$ . Clauses are essentially an alternative representation of a special class of such formulas, those in “conjunctive normal form”.)

There have been other attempts also to develop finite domain SAT solvers like ours. Frisch et al. [1], for example, developed a finite domain SAT solver, called NB-Walksat by extending Walksat, a hill-climbing algorithm used to solve Boolean SAT problems. However from their results it appears that their solver is not able to solve problems involving more than few hundred clauses. The largest we could find had 307 clauses.

Another example of closely related work is a multi-valued SAT solver called CAMA, presented by C. Liu et al. [2] that extends speed-up techniques used in state-of-the-art Boolean solvers and also introduces a novel technique, called *minimum value set (MVS)*, to improve the efficiency of conflict-based learning. (Conflict-based learning is a technique used by SAT solvers to improve the search strategy based on the past experience.) The input language of CAMA is different from ours only slightly, in that its atom has a range of values instead of just one. For example, instead of having  $c=v$  it could have  $c=v_1 \text{ OR } v_2 \text{ OR } v_3$ . However, it appears that much of the effort in developing CAMA has been diverted toward the more expressive feature of the language.

Also lots of research has been done in recent years on constraint satisfaction problem (CSP). Like SAT, a CSP consists of a finite set of variables, a domain of possible values of each variable, and a finite set of constraints. Each constraint restricts the combination of values that the set of variables may take. A solution to a CSP is an assignment of values to variables such that all the constraints are satisfied. In practice there are a wide variety of languages in which these constraints are expressed. In some CSP languages, finite domain SAT can be expressed quite nicely, but as far as we know, their solvers are not optimized for solving finite domain SAT problems, and it remains impractical to solve finite domain SAT problems by reduction to a

CSP language. Often the CSP languages are in some ways more expressive than finite domain theories – typically geared toward handling arithmetic equalities and inequalities – yet typical CSP languages cannot concisely represent finite domain theories. As far as we know, there is no CSP solver that is competitive on finite domain SAT problems.

## **1.5 Outline of the Thesis**

The rest of the thesis is organized as follows. Chapter 2 describes translations that convert a finite domain theory to the language of Sinha’s solver (by eliminating negation) and also to a Boolean theory. We used these translations in our experiments to compare the performance of our solver with various other solvers, and to demonstrate the computational advantages of our language. Chapter 3 describes the finite domain DPLL algorithm, as generalized in this thesis. Chapter 4 gives details of the implementation. Chapter 5 presents experimental results. Finally, Chapter 6 concludes the thesis and suggests future work.

## 2 Translations of Finite Domain Theories

In this chapter we describe three translations: one converts a finite domain theory to the language of Sinha's solver by eliminating negative literals, and two convert a finite domain theory to a Boolean theory.

### 2.1 Eliminating Negation

Sinha's finite domain SAT solver does not support negation i.e. it cannot work with literals of form  $c \neq v$ . But it is straightforward to eliminate negative literals from a finite domain theory. We simply replace each negative literal  $c \neq v$  with the atoms  $c = v'$  for every  $v' \in \text{dom}(c)$  such that  $v' \neq v$ . For example, if  $\text{dom}(A) = \{1, 2, 3\}$ , the clause  $\{A \neq 1\}$  would become  $\{A = 2, A = 3\}$ . We repeat this for all the negative literals in all the clauses. The following pseudo code defines the translation.

For every clause  $C$

For every negative literal  $c \neq v$  in  $C$

$$C := C - \{c \neq v\}$$

$$C := C \cup \{c = v' : v' \in \text{dom}(c), v' \neq v\}$$

For example, the clauses in (1) will be translated to

$$\{A=1, A=2, B=2\},$$

$$\{A=0, B=1\},$$

$$\{A=1, B=0, B=1, C=2\},$$

$$\{A=2, B=0, B=2\},$$

$$\{A=1, B=0, B=2\}.$$

The following well known fact states the correctness of this translation.

**Fact:** “Let  $T$  be a finite domain theory (possibly with negation) and  $T'$  be the corresponding translated finite domain theory with negative literals compiled out. An interpretation  $I$  is a model of  $T$  iff  $I$  is a model of  $T'$ .”

This thesis gives experimental evidence that it can be advantageous to solve finite domain theories with negation directly rather than compiling negation out.

## 2.2 Boolean Translations

Here we describe two translations from finite domain SAT to Boolean SAT. We call the first *quadratic translation* and the second *linear translation* (We explain why shortly). These translations are adapted from Appendix A of [4].

### 2.2.1 Quadratic Translation

In this translation, every atom  $c=v$  in a finite domain theory is understood as a Boolean atom i.e. as a symbol that interpretations will map into  $\{\mathbf{t},\mathbf{f}\}$ . Every occurrence of each finite domain atom  $c=v$  is “replaced” with the Boolean atom  $c=v$  and every occurrence of each finite domain literal  $c\neq v$  is replaced with the Boolean literal  $\neg c=v$ . In addition, we need clauses that will, roughly speaking, enforce the fact that a constant  $c$  takes exactly one value  $v$  in its domain. That is, for every constant  $c$ , these clauses will let Boolean atom  $c=v$  be true for exactly one  $v \in \text{dom}(c)$ . Thus, for every constant  $c$  we add (2) and (3) below to the theory.

$$\{c=v \mid v \in \text{dom}(c)\} \tag{2}$$

$$\{\neg c=v_1, \neg c=v_2\} \quad v_1, v_2 \in \text{dom}(c), v_1 \neq v_2 \tag{3}$$

Clause (2) enforces that at least one  $c=v$  is true and clauses (3) allow at most one  $c=v$  to be true. Thus they together make sure that exactly one  $c=v$  is true for every constant  $c$ . The number of extra clauses introduced in this translation is quadratic in the size of the domains; hence the name.

**Example** – Let’s take the example of the finite domain theory (1). Here we have three constants, each with domain  $\{0,1,2\}$ . Each member in the following list is now understood as a Boolean atom:

$A=0, A=1, A=2, B=0, B=1, B=2, C=0, C=1$  and  $C=2$ . The translated theory is shown below. Clauses (4) are directly generated from the theory, and clauses (5) and (6) are the additional clauses that enforce the fact that every constant takes exactly one value in its domain.

$$\{\neg A=0, B=2\}, \{A=0, B=1\}, \{A=1, \neg B=2, C=2\}, \{A=2, \neg B=1\}, \{A=1, \neg B=1\} \quad (4)$$

$$\{A=0, A=1, A=2\}, \{B=0, B=1, B=2\}, \{C=0, C=1, C=2\} \quad (5)$$

$$\{\neg A=0, \neg A=1\}, \{\neg A=0, \neg A=2\}, \{\neg A=1, \neg A=2\}$$

$$\{\neg B=0, \neg B=1\}, \{\neg B=0, \neg B=2\}, \{\neg B=1, \neg B=2\} \quad (6)$$

$$\{\neg C=0, \neg C=1\}, \{\neg C=0, \neg C=2\}, \{\neg C=1, \neg C=2\}$$

The following well known fact states the correctness of this translation.

For each interpretation  $I$  of the constants of the finite domain theory  $T$ , let  $I_Q$  be the corresponding Boolean interpretation such that, for all constants  $c$  of the language of  $T$  and all values  $v \in \text{dom}(c)$ ,  $I(c) = v$  iff  $I_Q(c=v) = \mathbf{t}$ .

**Fact:** “Let  $T$  be a finite domain theory and  $T'$  be its quadratic translation. An interpretation  $I$  is a model of  $T$  iff  $I_Q$  is a model of  $T'$ . Moreover, every model of  $T'$  can be written in the form of  $I_Q$ , for some interpretation  $I$  of  $T$ .”

### 2.2.2 Linear Translation

The quadratic translation is nice because of its simplicity. But it may become computationally unattractive when the domain sizes are large, since the number of additional clauses required is quadratic in the size of domains. In this section, we describe an alternative translation, in which the number of extra clauses is instead linear. This translation is also adapted from Appendix A of [4].

Notice that the quadratic cost of the previous translation is due to the clauses (3) that, roughly speaking, enforce that each finite domain constant gets at most one value from its domain. Hence the idea of the linear translation is to encode this constraint more concisely. To this end, for every constant  $c$  with  $\text{dom}(c) =$

$\{v_1, v_2, \dots, v_n\}$ , we introduce  $n - 1$  new Boolean variables,  $a_1, \dots, a_{n-1}$ . Then to enforce the fact that a constant can take at most one value from its domain, we add the following clauses to the theory.

$$\begin{aligned}
& \{\neg a_1, c=v_1\}, \{a_1, \neg c=v_1\}, \{\neg a_1, \neg c=v_2\} \\
& \{\neg a_2, a_1, c=v_2\}, \{a_2, \neg a_1\}, \{a_2, \neg c=v_2\}, \{\neg a_2, \neg c=v_3\} \\
& \dots \\
& \{\neg a_{n-2}, a_{n-3}, c=v_{n-2}\}, \{a_{n-2}, \neg a_{n-3}\}, \{a_{n-2}, \neg c=v_{n-2}\}, \{\neg a_{n-2}, \neg c=v_{n-1}\} \\
& \{\neg a_{n-1}, a_{n-2}, c=v_{n-1}\}, \{a_{n-1}, \neg a_{n-2}\}, \{a_{n-1}, \neg c=v_{n-1}\}, \{\neg a_{n-1}, \neg c=v_n\}
\end{aligned} \tag{7}$$

Consider an arbitrary constant  $c$ . The clause (2) guarantees that at least one of  $c=v_1, \dots, c=v_n$  is true. Let's show that (7) guarantees that at most one of these atoms is true. Assume that  $c=v_i$  is true. It follows from (7) that

- all of  $a_1, \dots, a_{i-1}$  must be false (due to  $\{\neg a_{i-1}, \neg c=v_i\}$  and  $\{a_{i-1}, \neg a_{i-2}\}, \dots, \{a_2, \neg a_1\}$ ), and
- all of  $a_i, \dots, a_{n-1}$  must be true (due to  $\{a_i, \neg c=v_i\}$  and  $\{a_{i+1}, \neg a_i\}, \dots, \{a_{n-1}, \neg a_{n-2}\}$ ),

and consequently that,

- all of  $c=v_1, \dots, c=v_{i-1}$  must be false (due to  $\{a_1, \neg c=v_1\}, \dots, \{a_{i-1}, \neg c=v_{i-1}\}$ ), and
- all of  $c=v_{i+1}, \dots, c=v_n$  must be false (due to  $\{\neg a_i, \neg c=v_{i+1}\}, \dots, \{\neg a_{n-1}, \neg c=v_n\}$ ).

Thus, we see that clauses (7) indeed make sure that for every constant  $c$  in the theory,  $c=v$  is true for at most one  $v \in \text{dom}(c)$ . In this translation the number of extra clauses introduced is linear in the size of the domains; hence the name.

**Example** – Let's again take the example of theory (1). Here we have three constants, each with domain  $\{0,1,2\}$ . As before, each atom of the finite domain theory is now understood as a Boolean atom. We will also introduce new atoms  $A_1, A_2, B_1, B_2, C_1, C_2$  – the auxiliary atoms required in clauses (7). The translated

theory will consist of clauses (4) and (5), along with the following instances of (7).

$$\begin{aligned}
& \{\neg A_1, A=0\}, \{A_1, \neg A=0\}, \{\neg A_1, \neg A=1\} \\
& \{\neg A_2, A_1, A=1\}, \{A_2, \neg A_1\}, \{A_2, \neg A=1\}, \{\neg A_2, \neg A=2\} \\
& \{\neg B_1, B=0\}, \{B_1, \neg B=0\}, \{\neg B_1, \neg B=1\} \\
& \{\neg B_2, B_1, B=1\}, \{B_2, \neg B_1\}, \{B_2, \neg B=1\}, \{\neg B_2, \neg B=2\} \\
& \{\neg C_1, C=0\}, \{C_1, \neg C=0\}, \{\neg C_1, \neg C=1\} \\
& \{\neg C_2, C_1, C=1\}, \{C_2, \neg C_1\}, \{C_2, \neg C=1\}, \{\neg C_2, \neg C=2\}
\end{aligned}$$

The only model  $I_L$  of the translated theory is as defined below.

$$\begin{aligned}
I_L(A=0) &= \mathbf{t}, I_L(A=1) = \mathbf{f}, I_L(A=2) = \mathbf{f} \\
I_L(B=0) &= \mathbf{f}, I_L(B=1) = \mathbf{f}, I_L(B=2) = \mathbf{t} \\
I_L(C=0) &= \mathbf{f}, I_L(C=1) = \mathbf{f}, I_L(C=2) = \mathbf{t} \\
I_L(A_1) &= \mathbf{t}, I_L(A_2) = \mathbf{t} \\
I_L(B_1) &= \mathbf{f}, I_L(B_2) = \mathbf{f} \\
I_L(C_1) &= \mathbf{f}, I_L(C_2) = \mathbf{f}
\end{aligned}$$

As mentioned earlier, the only model  $I$  of (1) is as follows.

$$I(A) = 0, I(B) = 2, I(C) = 2$$

We notice here that, for every atom  $c=v$ ,  $I(c) = v$  if and only if  $I_L(c=v) = \mathbf{t}$ .

Below we state the correctness of this translation.

For each interpretation  $I$  of the constants of the finite domain theory  $T$ , let  $I_L$  be the corresponding Boolean interpretation such that, for all constants  $c$  of the language of  $T$  and all values  $v \in \text{dom}(c)$ , where  $\text{dom}(c) = \{v_1, v_2, \dots, v_n\}$ ,  $I(c) = v_i$  iff  $I_L(c=v_i) = \mathbf{t}$ , and if  $I(c) = v_i$ , then for all  $j \in \{1, 2, \dots, n-1\}$

- 1)  $I_L(a_j) = \mathbf{f}$  if  $j < i$ , and

2)  $I_L(a_j) = \mathbf{t}$  if  $j \geq i$ .

**Fact:** “Let  $T$  be a finite domain theory and  $T'$  be its linear translation. An interpretation  $I$  is a model of  $T$  iff  $I_L$  is a model of  $T'$ . Moreover, every model of  $T'$  can be written in the form of  $I_L$ , for some interpretation  $I$  of  $T$ .”

**Improvement** To make the translation even more economical, for any constant  $c$  with a two element domain  $\{v_1, v_2\}$ , instead of adding any extra clauses or variables, we just replace in the finite domain theory:

- every occurrence of  $c=v_1$  or  $c \neq v_2$  with  $c=v_1$ , and
- every occurrence of  $c=v_2$  or  $c \neq v_1$  with  $\neg c=v_1$ .

In (1), for example, had the domain of  $B$  been  $\{1,2\}$ , (1) would have been translated to the following clauses, in addition to clauses (4).

$$\begin{aligned} & \{A=0, A=1, A=2\}, \{C=0, C=1, C=2\} \\ & \{\neg A_1, A=0\}, \{A_1, \neg A=0\}, \{\neg A_1, \neg A=1\} \\ & \{\neg A_2, A_1, A=1\}, \{A_2, \neg A_1\}, \{A_2, \neg A=1\}, \{\neg A_2, \neg A=2\} \\ & \{\neg C_1, C=0\}, \{C_1, \neg C=0\}, \{\neg C_1, \neg C=1\} \\ & \{\neg C_2, C_1, C=1\}, \{C_2, \neg C_1\}, \{C_2, \neg C=1\}, \{\neg C_2, \neg C=2\} \end{aligned}$$

### 2.2.3 Remarks on Boolean Translations

Sinha [25] showed that in principle it is probably better to solve finite domain SAT problems directly, rather than translating into Boolean SAT problems. First she showed, not surprisingly, that her finite domain solver is much faster on finite domain theories than on their Boolean translations. Second, she showed that her finite domain solver is generally faster solving finite domain problems than a competitive Boolean solver, SATO [14], working on the Boolean translations of the problems. (Interestingly SATO’s performance on quadratic translations was better than its performance on linear translations.) Third, Sinha showed that her

solver is comparable to the current state-of-the-art Boolean solver zChaff [20], when the quadratic translation is used. On the other hand, her solver cannot come close to zChaff's performance when the linear translation is used.

Essentially the same remarks apply to the solver introduced in the current thesis.

### 3 Finite Domain DPLL

In this chapter we describe our finite domain DPLL algorithm, an extension of Sinha’s finite domain DPLL to handle negative literals. The algorithm is essentially a depth-first heuristic search for a model, in the search space of *partial interpretations* (discussed next).

#### 3.1 Partial Interpretation

A *partial interpretation* is a set of literals, which for any given constant  $c$ ,

- (1) contains at most one element of  $\{c=v \mid v \in \text{dom}(c)\}$ , and
- (2) does not contain all elements of  $\{c \neq v \mid v \in \text{dom}(c)\}$ .

A partial interpretation  $P$  *satisfies* an atom  $c=v$  if  $c=v \in P$  and *satisfies* a literal  $c \neq v$  if either  $c \neq v \in P$  or  $c=v' \in P$  for some  $v' \neq v$ . Also we say  $P$  *falsifies* a literal if it satisfies its complement. (The complement of  $c=v$  is  $c \neq v$ ; the complement of  $c \neq v$  is  $c=v$ . If  $l$  is a literal, we sometimes write  $\bar{l}$  to denote its complement.) A partial interpretation *satisfies* a clause if it satisfies at least literal in it, and it *satisfies* a theory if it satisfies all the clauses in the theory. A partial interpretation *falsifies* a clause if it falsifies all literals in it, and it *falsifies* a theory if it falsifies any clause in the theory.

Whenever  $c=v \in P$ , intuitively we understand it as the value  $v$  is assigned to constant  $c$ . Thus with condition (1) above, we make sure that at most one value is assigned to any constant by  $P$ . Similarly, when  $c \neq v \in P$ , we understand it intuitively as the value  $v$  cannot be assigned to constant  $c$  i.e.  $c$  is prohibited from being assigned  $v$ . Condition (2) therefore guarantees that there is always at least one value  $v \in \text{dom}(c)$  that can be assigned to  $c$ . Thus the conditions maintain the “consistency” of partial interpretations.

Each partial interpretation  $P$  can be understood to correspond to the nonempty set of interpretations that satisfy all the literals in  $P$ . We call these interpretations the *extensions* of  $P$ . Notice that  $P$  satisfies a theory  $T$  if and only if all extensions of  $P$  satisfy  $T$ . Consequently  $T$  is satisfiable if and only if some partial interpretation satisfies  $T$ . Finally, notice that a partial interpretation  $P$  falsifies a theory  $T$  if and only if no extension of  $P$  satisfies  $T$ .

### 3.2 Depth-first Search

We will first describe depth-first search in the space of partial interpretations. Here we explore the space of partial interpretations, until we either find a partial interpretation that satisfies the theory or until we determine that no such partial interpretation exists. Figure 1 shows the pseudocode of the algorithm. Theory  $T$  and partial interpretation  $P$  are the input parameters. Initially  $P$  is empty. It first checks if  $P$  satisfies or falsifies  $T$ . If  $P$  does neither, it chooses a literal  $l$  such that  $P \cup \{l\}$  is a partial interpretation. Such a literal  $l$  is guaranteed to exist as explained below.

At this point, since  $P$  neither satisfies nor falsifies  $T$ , there is at least one clause  $C \in T$ , which is neither satisfied nor falsified by  $P$  which means there is at least one literal  $l \in C$  which is neither satisfied nor falsified by  $P$ . Literal  $l$  is either a negative literal  $c \neq v$  or an atom  $c = v$ . In either case we know that the atom  $c = v$  is neither satisfied nor falsified. This implies that for no  $v' \neq v$ ,  $c = v' \in P$ . Therefore, we can safely add  $c = v$  to  $P$ , without violating the restriction on atoms mentioned in the definition of a partial interpretation.

After adding  $l$  to  $P$ , the algorithm calls itself recursively with  $T$  and the extended partial interpretation  $P \cup \{l\}$ . If the call does not return a failure, it means the problem is solved. Otherwise, the algorithm tries to add the complement  $\bar{l}$  of  $l$  to  $P$  instead of  $l$ . If this still a valid partial interpretation, it makes another recursive call with the extended partial interpretation  $P \cup \{\bar{l}\}$  and returns its outcome. Otherwise it returns a failure. Notice that  $P \cup \{\bar{l}\}$  may not necessarily be a partial interpretation. (For example, if  $\bar{l}$  is  $c \neq v$  and for all  $v' \in \text{dom}(c)$ ,  $v' \neq v$ ,  $c \neq v' \in P$ , then  $P \cup \{\bar{l}\}$  won't be a partial interpretation, since it violates the restriction on negative literals.) Hence this check is necessary.

Let's look at the following theory with signature  $\{A, B\}$  and domains  $\text{dom}(A) = \text{dom}(B) = \{0, 1, 2\}$ .

$$\begin{aligned} &\{A=1\} \\ &\{A \neq 0, B=2\} \\ &\{A=0, B=0\} \end{aligned} \tag{8}$$

We start by picking a literal. Let's say we pick  $A=0$  and satisfy it i.e.  $P = \{A=0\}$ . The search tree

```

DF( $T, P$ )
  if  $P$  satisfies  $T$ , then return  $P$ 
  if  $P$  falsifies  $T$ , then return fail
  Choose a literal  $l$  such that  $P \cup \{l\}$  is a partial interpretation
   $P' := \text{DF}(T, P \cup \{l\})$ 
  if  $P' \neq \text{fail}$ , then return  $P'$ 
  if  $P \cup \{\bar{l}\}$  is not a partial interpretation, then return fail
  return  $\text{DF}(T, P \cup \{\bar{l}\})$ 

```

Figure 1: Pseudo code of DF Algorithm

constructed at this point is shown in Figure 2. The **t** on the branch indicates that  $A=0$  (the chosen literal) has been added to  $P$ , and so is satisfied along this branch. Notice that  $P$  at this point falsifies the only literal in the first clause. So that clause is falsified by  $P$ , and so the theory is falsified. We therefore have to backtrack and instead falsify the most recently satisfied literal,  $A=0$ . Satisfying  $A \neq 0$  satisfies clause 2.

Next let's say we pick literal  $A=1$  to satisfy. So at this point  $P = \{A \neq 0, A=1\}$ . This satisfies clause 1, as well. Now only clause 3 is unsatisfied. So again we pick a literal to satisfy. We cannot pick  $A=2$ , since it's

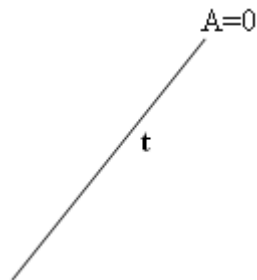


Figure 2: Search Tree

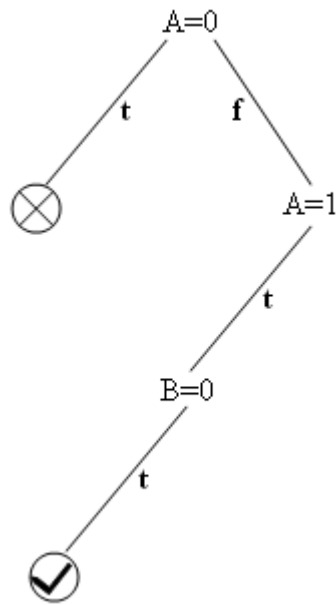


Figure 3: Final Search Tree

already falsified. So let's say we pick  $B=0$  and satisfy it. This satisfies clause 3, and since all three clauses are satisfied, the theory is satisfied. Final partial interpretation is  $\{A \neq 0, A=1, B=0\}$ . Final search tree is shown in Figure 3.

### 3.3 Reduction of Theory

During the search, whenever a clause is satisfied we can ignore it in the subsequent part of that branch, since that clause can no longer affect satisfaction of the theory. So it is convenient to think of it as “removed” from the theory. Similarly when a literal  $l$  is falsified, all its occurrences can be safely removed from the theory, since no extension of the current partial interpretation can satisfy that literal and so that literal can play no further role in the attempt (along that branch) to satisfy the theory. For example, in (8) after we falsified  $A=1$ , we could not have subsequently satisfied clause 1 by satisfying  $A=1$ , since it was already falsified. We therefore think of all such literals as “removed” from their respective clauses and the clauses are said to have

been “reduced”. Thus when all the literals in a clause are falsified, the clause becomes “empty”. Of course an empty clause is unsatisfiable, so when the empty clause is encountered, this indicates an inconsistency in the assignment; it is time to backtrack in the depth first search. On the other hand, if ever we encounter the empty theory, we have succeeded in satisfying the theory.

Thus to summarize, when an atom  $c=v$  is satisfied, the theory is reduced by

- 1) removing from the theory all clauses that contain  $c=v$ , and
- 2) removing from their respective clauses all occurrences of the falsified literals  $c\neq v$ , and  $c=v'$  for all  $v' \in \text{dom}(c)$  such that  $v' \neq v$ .

Similarly, when  $c\neq v$  is satisfied, the theory is reduced by

- 1) removing from the theory all clauses that contain  $c\neq v$ , and
- 2) removing from their respective clauses all occurrences of the falsified literal  $c=v$ .

Or, to put it another way, when  $P$  neither satisfies nor falsifies  $T$ , one can answer the question whether some extension of  $P$  satisfies  $T$  by considering a reduced version of  $T$ , in which all clauses satisfied by  $P$  are removed from  $T$  and all remaining occurrences of literals falsified by  $P$  are removed.

Figure 4 shows a revision of the DF algorithm in which the idea of reduction is implemented. Reduce routines implement the above mentioned reduction scheme for positive and negative literals.

### 3.4 Safe and Obvious Assignments

Brute force depth-first search can be improved by taking better advantage of the structure of the given finite domain theory. In (8), for example, the first clause contained only one atom. So if we falsify it, the clause will become empty, which is what happened when we tried to satisfy  $A=0$  in the previous example. Therefore instead of spending time on assignments that will immediately yield the empty clause, we can make obvious assignments, such as satisfying  $A=1$ , first. This is exactly what is done in finite domain DPLL. In addition,

DFR( $T, P$ )

if  $T = \emptyset$  then return  $P$

if  $\emptyset \in T$  then return fail

Choose a literal  $l$  such that  $P \cup \{l\}$  is a partial interpretation

$P' := \text{DFR}(\text{Reduce}(T, l), P \cup \{l\})$

if  $P' \neq \text{fail}$ , then return  $P'$

if  $P \cup \{\bar{l}\}$  is not a partial interpretation, then return fail

return  $\text{DFR}(\text{Reduce}(T, \bar{l}), P \cup \{\bar{l}\})$

Reduce( $T, c \neq v$ )

return  $\{C - \{c=v\} \mid C \in T, c \neq v \notin C\}$

Reduce( $T, c=v$ )

return  $\{C - (\{c \neq v\} \cup \{c=v' \mid v' \in \text{dom}(c), v' \neq v\}) \mid$

$C \in T, c=v \notin C, C \cap \{c \neq v' \mid v' \in \text{dom}(c), v' \neq v\} = \emptyset\}$

Figure 4: Pseudo code of DFR Algorithm

there are two more mechanisms in finite domain DPLL that detect such safe and obvious assignments. We discuss all three in this section.

### 3.4.1 Unit Clause Propagation (UCP)

A *unit clause* is a clause that has only one literal in it. The only way to satisfy such a clause is to satisfy the literal in it. Every partial interpretation that satisfies the theory, and so the clause, has to satisfy this literal. Therefore, we are sure that in satisfying this literal we cannot go “wrong”. So we satisfy the literal in it and grow the partial interpretation. This may in turn, after reduction, give rise to more unit clauses; hence the name *Unit Clause Propagation*. It may also generate empty clauses (at which point we know that no extension of the current partial interpretation can satisfy the theory).

To understand UCP better, let’s look at the following example. The signature is  $\{A, B, C\}$  and domains are  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ . Here is the theory.

Clause 1:  $\{A=1\}$

Clause 2:  $\{A \neq 2, B=2\}$

Clause 3:  $\{A=3, B \neq 2\}$

Clause 4:  $\{B=2, C \neq 2\}$

Clause 5:  $\{B=2, C \neq 0\}$

Clause 1 is a unit clause,  $A=1$  being the only literal (atom) in it. So we satisfy it, and as a result the theory reduces to:

Clause 1: “Satisfied”

Clause 2: “Satisfied” (Due to  $A \neq 2$ )

Clause 3:  $\{B \neq 2\}$  ( $A=3$  removed)

Clause 4:  $\{B=2, C \neq 2\}$  ....(Unaffected)

Clause 5:  $\{B=2, C \neq 0\}$  ....(Unaffected)

In the reduced theory Clause 3 is a unit clause. So we satisfy its literal,  $B \neq 2$ , and reduce the theory to:

Clause 1: “Satisfied”

Clause 2: “Satisfied”

Clause 3: “Satisfied”

Clause 4:  $\{C \neq 2\}$  ( $B=2$  removed)

Clause 5:  $\{C \neq 0\}$  ( $B=2$  removed)

Both the clauses left are unit clauses now. They are handled one by one. Satisfying one does not affect the other. They can be satisfied in any order and the whole theory is satisfied.

Now if we take a closer look at constant  $C$ , we realize that out of the 3 values in its domain  $\{0,1,2\}$ , two possibilities – 0 and 2 – are already ruled out when we satisfy Clause 4 (i.e. literal  $C \neq 2$ ) and Clause 5 (i.e. literal  $C \neq 0$ ). This means that  $C$  now must be assigned 1 i.e. we know that we would not be “wrong” if we made  $C=1$  true. This is what is called “*Entailment*” (discussed next) and the atom  $C=1$  is said to be *entailed*.

### 3.4.2 Entailment Handling

Let’s now define the notion of entailment more formally. During the process of exploring the search space, for some constant  $c$  and some value  $v \in \text{dom}(c)$ , if  $c=v \notin P$  but  $c=v' \in P$  for all  $v' \in \text{dom}(c)$  such that  $v' \neq v$ , then the atom  $c=v$  is said to be entailed by  $P$ . All the entailed atoms are satisfied to advance the search.

### 3.4.3 Pure Literal Handling

A *pure literal*, roughly speaking, is a literal which, when satisfied, only satisfies some clauses without falsifying any. There are two types of pure literals, positive and negative.

(A) Positive Pure Literal (Pure Atom): The literal  $c=v$  is pure if the following three conditions hold.

- i. For all  $v' \in \text{dom}(c)$  and  $v' \neq v$ ,  $c=v'$  does not occur in the theory.

- ii.  $c \neq v$  does not occur in the theory.
- iii. Either
  - (a)  $c = v$  occurs in the theory, or
  - (b) for at least one  $v' \in \text{dom}(c)$  and  $v' \neq v$ ,  $c \neq v'$  occurs in the theory.

When an atom  $c = v$  is satisfied, we know that all occurrences of atoms  $c = v'$  ( $v' \neq v$ ) and of literal  $c \neq v$  are removed from the theory. Conditions i and ii make sure that these do not occur in the theory. So no literal is removed from any clause. Condition iii makes sure that when this literal is satisfied, at least one clause is satisfied. In other words the condition makes the pure atom is “useful”.

Let’s look at the following example. The signature is  $\{A, B, C\}$  and the domains are  $\text{dom}(A) = \text{dom}(B) = \text{dom}(C) = \{0, 1, 2, 3\}$ . The theory has two clauses:

$$\{A=1, B \neq 2, C=1\}, \{A=3, B=2, C \neq 2\}.$$

The atom  $C=1$  is pure: it satisfies all the conditions mentioned above. Satisfying it satisfies both the clauses.

(B) Negative Pure Literal: The literal  $c \neq v$  is pure if –

- i. for all  $v' \in \text{dom}(c)$ ,  $c = v'$  is not a positive pure literal,
- ii.  $c = v$  does not occur in the theory, and
- iii.  $c \neq v$  occurs in the theory.

Condition i ensures that there are no positive pure literals on this constant. This is because, if there are pure atoms on the constant, it is better to satisfy them first. (The example below demonstrates this.)

Condition ii guarantees that nothing is falsified, and condition iii makes sure that the picked literal is “useful”.

In the following example the signature is  $\{A, B, C\}$  and the domains are  $\text{dom}(A) = \text{dom}(B) = \text{dom}(C) = \{0, 1, 2, 3\}$ . Here is the theory:

$$\{A=1, B \neq 2, C=1\}, \{A=3, B=2, C \neq 2\}, \{A \neq 2, B=3, C \neq 3\}.$$

Here  $A \neq 2$  is a negative pure literal. But  $C \neq 2$  and  $C \neq 3$  are not, because we already have a positive pure literal on  $C$ , namely  $C=1$ . When  $C=1$  is satisfied, that also satisfies  $C \neq 2$  and  $C \neq 3$ . But if we ignored this fact and satisfied  $C \neq 2$  and/or  $C \neq 3$  before  $C=1$ , then although we are not doing anything wrong, we are certainly ignoring something stronger and more useful.

Satisfying a pure literal never results in generation of empty clauses or unit clauses; it only satisfies some clauses and does not affect any others.

### 3.5 Finite Domain DPLL – Algorithm

Now we have all the background we need, so let's look at the finite domain DPLL algorithm. This algorithm is essentially an addition of safe and obvious assignment handling mechanisms onto the DFR algorithm. Figure 5 shows pseudo code of the algorithm. Notice that unlike DFR, here we do not perform the consistency check before adding  $\bar{l}$  to  $P$ . This is because in DFR there is no other mechanism to ensure that not all  $c \neq v$ ,  $v \in \text{dom}(c)$  are added to  $P$ . On the other hand, in finite domain DPLL, the entailment handling mechanism adds atom  $c=v$  to  $P$ , the moment all  $c \neq v'$  ( $v' \in \text{dom}(c)$  and  $v' \neq v$ ) are added to  $P$ , thereby falsifying  $c \neq v$ , the only literal that could have caused the inconsistency. Since this is taken care of before the literal picking, the check becomes unnecessary.

### 3.6 Heuristics

In this thesis we experimented with five different heuristics for choosing a literal to branch on.

#### 3.6.1 Heuristic 1: Highest Satisfaction Under Maximum Impact

It will help to define a few terms before we look into this heuristic.

*Impact* of a constant is the total number of unsatisfied clauses that contain at least one literal involving the constant. For example, if we have 5 clauses involving constant  $c$ , then  $c$ 's impact is 5.

```

DPLL( $T, P$ )
  if  $T = \emptyset$ , then return  $P$ 
  if  $\emptyset \in T$ , then return fail
  if there exists a unit clause  $\{l\} \in T$ ,
    then return DPLL(Reduce( $T, l$ ),  $P \cup \{l\}$ )
  if there exists an entailed atom  $c=v$  with respect to  $P$ ,
    then return DPLL(Reduce( $T, c=v$ ),  $P \cup \{c=v\}$ )
  if there exists a pure literal  $\{l\}$  in  $T$ ,
    then return DPLL(Reduce( $T, l$ ),  $P \cup \{l\}$ )
   $l := \text{pickNextLiteral}()$ 
   $P' := \text{DPLL}(\text{Reduce}(T, l), P \cup \{l\})$ 
  if  $P' \neq \text{fail}$ , then return  $P'$ 
  return DPLL(Reduce( $T, \bar{l}$ ),  $P \cup \{\bar{l}\}$ )

Reduce( $T, c \neq v$ )
  return  $\{C - \{c=v\} \mid C \in T, c \neq v \notin C\}$ 

Reduce( $T, c=v$ )
  return  $\{C - (\{c \neq v\} \cup \{c=v' \mid v' \in \text{dom}(c), v' \neq v\}) \mid$ 
     $C \in T, c=v \notin C, C \cap \{c \neq v' \mid v' \in \text{dom}(c), v' \neq v\} = \emptyset\}$ 

```

Figure 5: pseudo code of DPLL Algorithm

*Satisfaction* of a literal is the total number of clauses that will be satisfied when the literal is satisfied. For example, in the theory  $\{ \{A=0, B=1\}, \{A \neq 1, C=1\} \}$ , with signature  $\{A, B, C\}$  and domains  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ , the heuristic will pick  $A=0$  because impact of  $A$  is 2, which is more than any other constant and then it will pick value 0 for  $A$ , since the satisfaction of  $A=0$  is 2, which is more than any other  $A=v$ , where  $v \in \{1, 2\}$ . (When a value is picked for the chosen constant  $c$ , the heuristic does not consider negative literals  $c \neq v$  ( $v \in dom(c)$ ), because the satisfaction of every negative literal  $c \neq v$  is no greater than that of any atom  $c=v'$  such that  $v' \neq v$  and  $v' \in dom(c)$ . Hence the best candidate is still guaranteed by considering just the atoms.) Thus the heuristic is always guaranteed to affect maximum number of clauses and then to satisfy as many as possible.

### 3.6.2 Heuristic 2: Highest (Satisfaction - Falsification)

*Falsification* of a literal is the total number of unsatisfied clauses that will be “reduced” when the literal is satisfied. This heuristic computes the value (*satisfaction - falsification*) for every literal in the theory and picks one so as to maximize the above value. The idea behind this heuristic is that the picked literal will not only satisfy a high number of clauses (due to the term *satisfaction*) but will also keep the number of reduced clauses low (due to the term  $-falsification$ ) thereby making subsequent backtracks less likely. (The more clauses we reduce, the more likely it becomes to generate an empty clause.)

Let's look the following theory with signature  $\{A, B, C\}$  and domains  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ .

$$\{A=0, B=1\}, \{A \neq 1, C=1\}, \{A=1, B \neq 0\} \quad (9)$$

The heuristic will pick  $B=1$  instead of  $A=0$ , although the value of *satisfaction* for both is 2. This is because the value of *falsification* is 1 for  $A=0$  and 0 for  $B=1$ . Consequently, the value (*satisfaction - falsification*) comes to  $2 - 1 = 1$  for  $A=0$  and  $2 - 0 = 2$  for  $B=1$ .

Thus the heuristic tries to satisfy as many clauses as possible and at the same time minimize the number of shrinking clauses.

### 3.6.3 Heuristic 3: Highest (Satisfaction - Falsification) Negative Literals Only

When we make an atom true we are making a strong statement about the constant in the atom. We are essentially assuming that constant has that value. This increases the possibility of backtracking. Therefore in this heuristic we used the same criterion as in Heuristic 2 but restricted the search to only negative literals. For example, in theory (9) there are two negative literals, namely  $A \neq 1$  and  $B \neq 0$ . The heuristic picks  $B \neq 0$ , because the value (*satisfaction - falsification*) for it is 1 whereas that for  $A \neq 1$  is 0. The heuristic does not consider any atoms.

### 3.6.4 Heuristic 4: Highest (Satisfaction - Falsification) Atoms Only

In this heuristic we explored the other half of the literals. Keeping the criterion the same, we considered only the atoms and picked the one with highest value of (*Satisfaction - Falsification*). Here although we are increasing the possibility of backtracking, we are shrinking the theory down faster. So whether a contradiction or success, we hope to find it quicker.

For example, let's look at the following theory, with signature  $\{A, B, C\}$  and domains  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ .

$$\{A=0, B \neq 1\} \quad \{A \neq 1, B \neq 1\} \quad \{C=1, B \neq 1\} \quad (10)$$

The heuristic will pick  $A=0$  with the value (*satisfaction - falsification*) = 2, even when we have a literal ( $B \neq 1$ ) whose corresponding value is 3, because it considers only atoms.

### 3.6.5 Heuristic 5: Highest Satisfaction

This heuristic picks the atom whose *satisfaction* is maximum. Here we do not consider negative literals, because for every negative literal  $c \neq v$ , there is always some atom  $c = v'$  ( $v' \neq v$ ), such that the value of *satisfaction* of  $c = v'$  is no less than that of  $c \neq v$ . Thus even if we ignore the negative literals, we are still guaranteed to get the best candidate without having to waste time considering negative literals. For

	0	1	2	
A	1	0	1	=
	0	0	4	≠

	0	1	2	
B	2	1	1	=
	1	1	1	≠

	0	1	2	
C	0	1	0	=
	0	1	0	≠

Figure 6: Example 1: Number of Clauses for Each Literal

example, in theory (10) the heuristic might pick  $B=0$ , even though it does not occur in the theory, because its *satisfaction* is 3, which is no less than that of any other atom.

### 3.7 Examples

Let's now look at some examples to see how the algorithm works. One of the reasons behind experimenting with five different heuristics is that no heuristic is "right" or "wrong". A heuristic may work better than another on a problem, while another may do better than the first on some other problem. We demonstrate that with the following examples. In these examples we compared and contrasted Heuristic 1 and Heuristic 5.

#### 3.7.1 Example 1:

Let's consider the example of the following theory where the signature is  $\{A, B, C\}$  and the domains are  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ .

$$\{A=2, B \neq 0\}, \{A \neq 2, B=1\}, \{B \neq 1, C=1\}$$

$$\{A \neq 2, B=2, C \neq 1\}, \{A=0, B=0\}, \{A \neq 2, B \neq 2\}, \{A \neq 2, B=0\}$$

Figure 6 shows the number of clauses each literal occurs in.

Using Heuristic 1 Impact of  $A$  is 6 and that of  $B$  is 7, so  $B$  wins over  $A$ . Now the heuristic picks the literal  $B=0$  because its satisfaction is 4, which is greater than any other  $B=v$  ( $v \in \{1,2\}$ ).  $B=0$  is satisfied and the theory reduces to

$$\{A=2\}, \{A \neq 2\}, \{A \neq 2, C \neq 1\}.$$

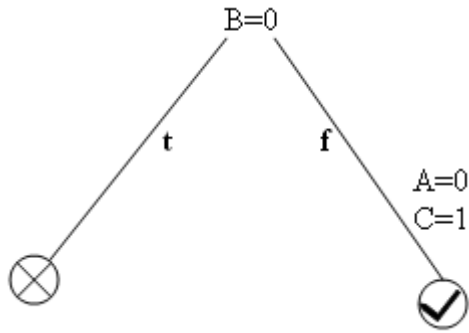


Figure 7: Search Tree - Heuristic 1 on Example 1

The routine is called recursively on the reduced theory. Notice that it has two unit clauses  $\{A=2\}$  and  $\{A\neq 2\}$ , which cannot both be satisfied. If one is satisfied the other becomes empty. Therefore this invocation returns fail (since its child invoked from its UCP mechanism returned fail). So back in the topmost invocation the most recently satisfied atom,  $B=0$ , is now falsified, the original theory is reduced one more time, and another recursive call is made with this reduced version of the theory, thereby opening another branch in the search tree. The new reduced theory is shown below.

$$\{A\neq 2, B=1\}, \{B\neq 1, C=1\},$$

$$\{A\neq 2, B=2, C\neq 1\}, \{A=0\}, \{A\neq 2, B\neq 2\}, \{A\neq 2\}$$

In the second level invocation, UCP detects  $\{A=0\}$  as a unit clause, satisfies it, reduces the theory further as shown below and invokes the routine on the reduced theory

$$\{B\neq 1, C=1\}$$

In the third level call,  $C=1$  is caught by the pure literal mechanism and is satisfied causing the theory to become empty. The problem is solved, the chain of calls winds up returning the current partial interpretation back to the topmost level. Extension of the current partial interpretation gives a model of the theory, one that satisfies  $A = 0$ ,  $B = 1$  and  $C = 1$ . Figure 7 shows the corresponding search tree.

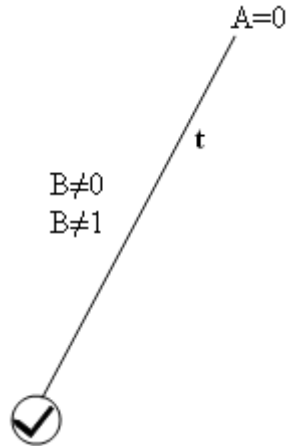


Figure 8: Search Tree - Heuristic 5 on Example 1

Using Heuristic 5 Now let's see how Heuristic 5 behaves on the above theory. Since there are no obvious or safe assignments in the given theory, the algorithm begins by picking a literal using Heuristic 5. It selects the literal  $A=0$ , because its *satisfaction* is 5, which is highest, and makes it **t**. The theory is reduced to:

$$\{B \neq 0\}, \{B \neq 1, C = 1\}$$

The routine is called recursively on the reduced theory, starting second level invocation. Notice that in the reduced theory  $\{B \neq 0\}$  is a unit clause. UCP satisfies it and makes third level call, which in turn detects  $B \neq 1$  as a negative pure literal and satisfies it. This satisfies the theory. The chain of calls winds up returning the current partial interpretation back to the topmost level. Extension of the current partial interpretation gives a model of the theory, one that satisfies  $A = 0$ ,  $B \neq 0$ , and  $B \neq 1$ . Figure 8 shows the corresponding search tree.

Thus with Heuristic 1 we had to backtrack once, but with Heuristic 5 we did not have to backtrack at all.

### 3.7.2 Example 2:

Let's now look at the following theory. Here also the signature is  $\{A, B, C\}$  and the domains are  $dom(A) = dom(B) = dom(C) = \{0, 1, 2\}$ .

$$\{A=0, C=1\}, \{A=2, B \neq 0\}, \{A=0, B=2\}, \{A=0, B=0\}$$

	0	1	2	
A	3	2	1	=
	1	1	1	

	0	1	2	
B	2	0	1	=
	1	0	4	

	0	1	2	
C	0	1	0	=
	0	1	0	

Figure 9: Example 2: Number of Clauses for Each Literal

$$\{A=1, B \neq 2\}, \{A=1, B \neq 2, C \neq 1\}, \{A \neq 1, B \neq 2\}, \{A \neq 0, B \neq 2\}, \{A \neq 2, B=0\}$$

Figure 9 shows the number of clauses each literal occurs in.

Using Heuristic 1 The impact of  $A$  is 9, which is the highest. So  $A$  is picked. Next the heuristic picks the literal  $A=0$  because its satisfaction is 5, which is greater than any other  $A=v$  ( $v \in \{1,2\}$ ).  $A=0$  is satisfied and the theory reduces to

$$\{B \neq 0\}, \{B \neq 2\}, \{B \neq 2, C \neq 1\}, \{B \neq 2\}$$

The routine is called recursively on the reduced theory. In the reduced theory  $\{B \neq 0\}$  and  $\{B \neq 2\}$  are unit clauses. They are captured and satisfied by second and third level of invocation. This satisfies the theory. The chain of calls winds up returning the current partial interpretation back to the topmost level. Extension of the current partial interpretation gives a model of the theory, one that satisfies  $A = 0$ ,  $B \neq 0$ , and  $B \neq 2$ . Figure 10 shows the corresponding search tree.

Using Heuristic 5 Now let's see how Heuristic 5 behaves on the above theory. Since there are no obvious or safe assignments in the given theory, the algorithm begins by picking a literal using Heuristic 5. It selects the literal  $B=0$ , because its *satisfaction* is 6, which is highest. The theory reduces to:

$$\{A=0, C=1\}, \{A=2\}, \{A=0\}.$$

The routine is called recursively on the reduced theory. Notice that it has two unit clauses  $\{A=2\}$  and  $\{A=0\}$ , which cannot both be satisfied. If one is satisfied the other becomes empty. Therefore this invocation returns fail (since its child invoked from its UCP mechanism returned fail). So back in the topmost invocation the

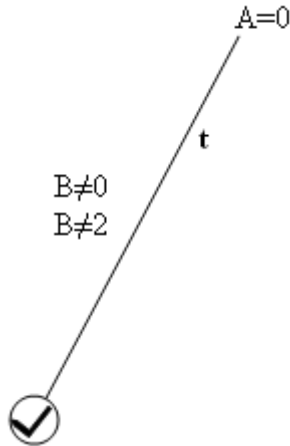


Figure 10: Search Tree - Heuristic 1 on Example 2

most recently satisfied atom,  $B=0$ , is now falsified, the original theory is reduced one more time, and another recursive call is made with this reduced version of the theory, thereby opening another branch in the search tree. The new reduced theory is shown below.

$$\{A=0, C=1\}, \{A=0, B=2\}, \{A=0\},$$

$$\{A=1, B\neq 2\}, \{A=1, B\neq 2, C\neq 1\}, \{A\neq 1, B\neq 2\}, \{A\neq 0, B\neq 2\}, \{A\neq 2\}$$

In the second level invocation, UCP detects  $\{A=0\}$  as a unit clause, satisfies it, reduces the theory further as shown below and invokes the routine on the reduced theory.

$$\{B\neq 2\}, \{B\neq 2, C\neq 1\}$$

In the third level call  $\{B\neq 2\}$  is caught by the UCP mechanism and is satisfied causing the theory to become empty. The problem is solved, and the chain of calls winds up returning the current partial interpretation back to the topmost level. Extension of the current partial interpretation gives a model of the theory, one that satisfies  $A = 0$  and  $B = 1$ . Figure 11 shows the corresponding search tree.

Thus in this example we notice that Heuristic 1 solved the problem without any backtracking but Heuristic 5 needed one backtrack. Thus in this case Heuristic 1 performed better than Heuristic 5, which is opposite to

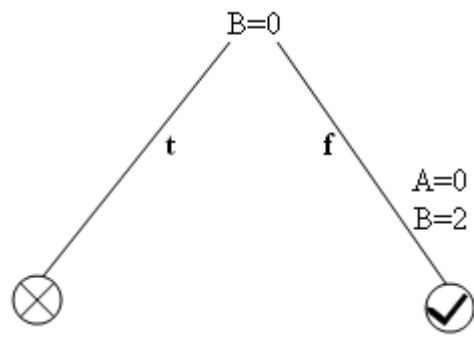


Figure 11: Search Tree - Heuristic 5 on Example 2

what we observed in Example 1.

## 4 Implementation

We began our implementation by using the data structures provided by the Standard Template Library (STL) of C++. After the first implementation was complete, we felt that these data structures were too “general purpose”, providing support for extraneous functions, with associated overhead. We therefore started over, implementing custom-tailored data structures that provided exactly the functionality we needed. With these data structures we observed that the speed was improved and larger problem instances were handled successfully. In this chapter we describe in detail all these data structures and also other challenges faced in the implementation of this solver. Readers uninterested in implementation details can safely skip this chapter.

In this algorithm we have seven main functionalities described below:

- (1) Empty theory detection
- (2) Empty clause detection
- (3) Unit clause detection
- (4) Entailment handling
- (5) Pure literal detection
- (6) Branching: computation of heuristic
- (7) Backtracking: w/o multiple copies of the theory

Before discussing how each of these is implemented, let's first understand the data structures.

### 4.1 Data Structures

We have five main classes described below.

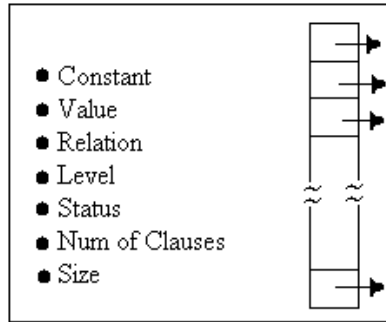


Figure 12: Literal

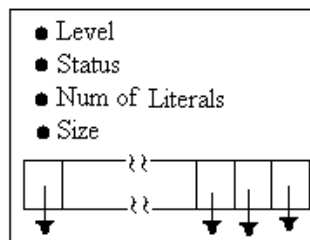


Figure 13: Clause

#### 4.1.1 Literal

This is one of the simplest classes to understand. Figure 12 shows some important elements of the class. It has a data member for each of these: constant (int), value (int), relation (Boolean), level (int), status (satisfied/deleted/default) and number of unsatisfied clauses this literal occurs in (int). It also has an array of pointers to Clauses (another class, discussed shortly), and a data member for size of the array (int).

#### 4.1.2 Clause

Figure 13 shows some important elements of this class. It keeps an array of pointers to the literals, size of the array (int), the number of default literals (int) occurring in this clause, status of the clause (satisfied or not) and level (int).

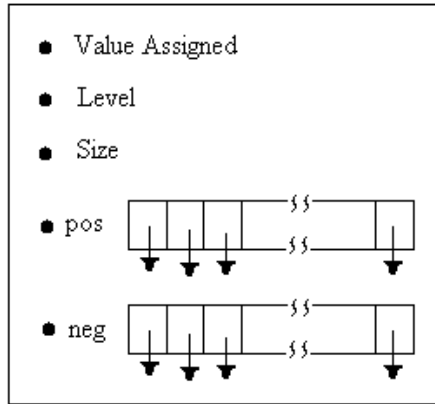


Figure 14: Literal Pool Node (LPN) Class

#### 4.1.3 Literal Pool Node (LPN)

This class represents a constant in the theory, meaning there is one object of this class for every constant in the signature. Its purpose is to hold all the literals on the constant  $c$  it represents. It classifies the literals into two categories, positive (i.e. of the form  $c=v$ ) and negative (i.e. of the form  $c\neq v$ ). For each category it maintains an array of pointers to literals. The arrays are named *pos* and *neg* and each has size  $dom(c)$ . As their names suggest *pos* holds all the positive literals whereas *neg* holds all the negative literals. Figure 14 shows the class. There are other data members also that are either essential (e.g. size of the arrays) or useful (e.g. value assigned, level).

#### 4.1.4 Literal Pool (LP)

Purpose of this class is to hold all the LPN objects. It, therefore, has an array (called *literalPool*) of LPN objects whose size is same as that of the signature (since there is one LPN object per constant in the signature).

Figure 15 shows the class.

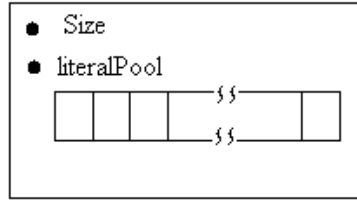


Figure 15: Literal Pool (LP) Class

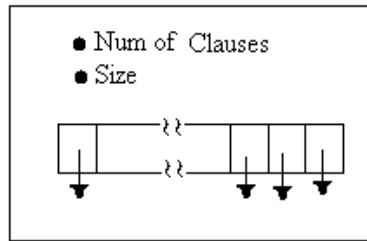


Figure 16: Set Class

#### 4.1.5 Set

Purpose of this class is to hold the clauses. Accordingly, it has an array of pointers to Clauses. It also has data members for size of the array and number of unsatisfied clauses. Figure 16 shows the class.

#### 4.1.6 Class Diagram

Figure 17 shows the class diagram. Literal and Clause are the lowest level classes and have associations with each other. Set is an aggregate of Clause. LPN is an aggregate of Literal and is aggregated in LP.

### 4.2 Example

Let's consider the following theory. Here the signature is  $\{1,2\}$  and the domains are  $dom(1) = dom(2) = \{0, 1, 2\}$ . Figure 18 shows how the data structures will look for this theory. Here is the theory:

Clause 1:  $\{1=0, 2=1\}$

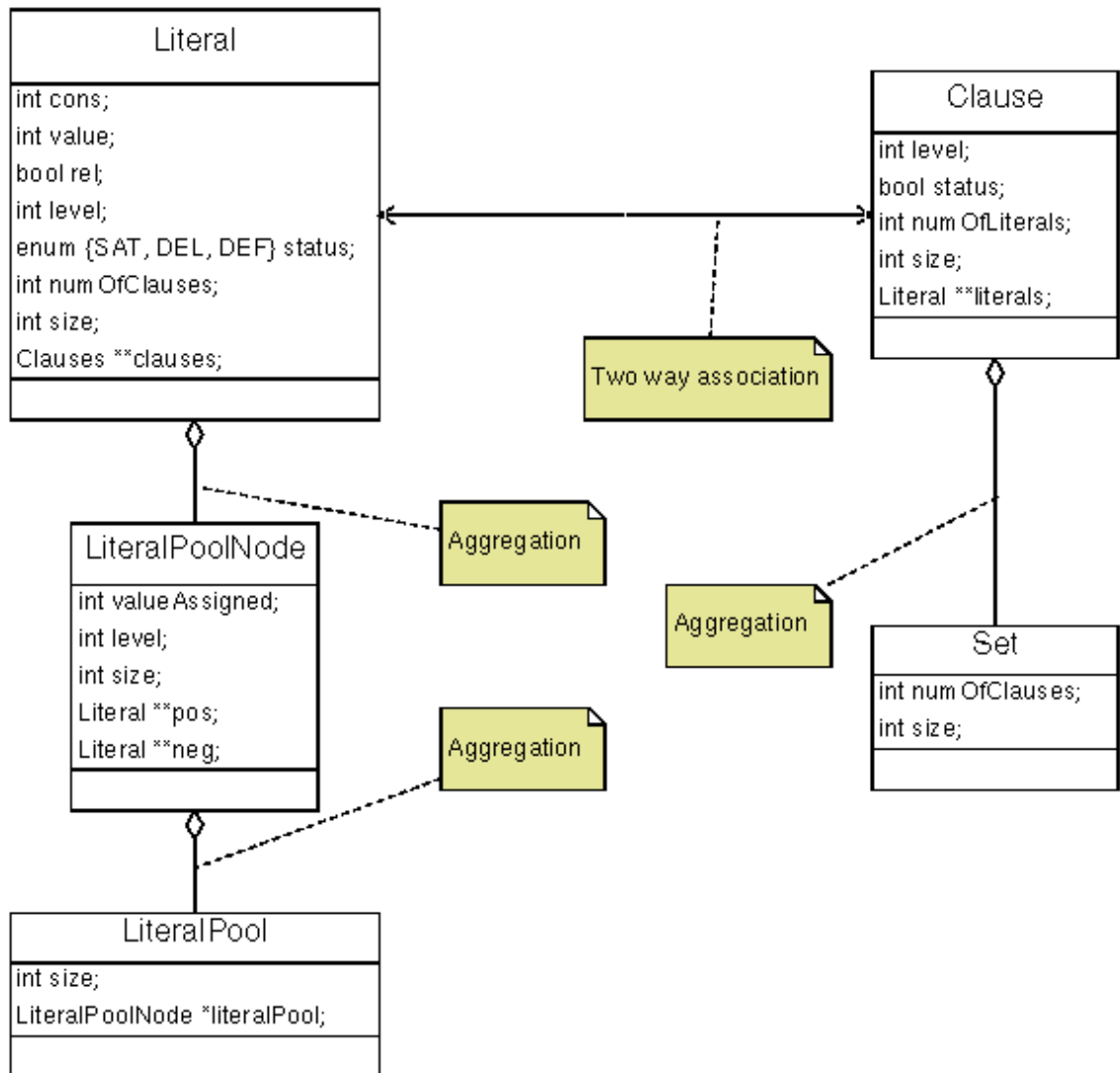


Figure 17: Class Diagram

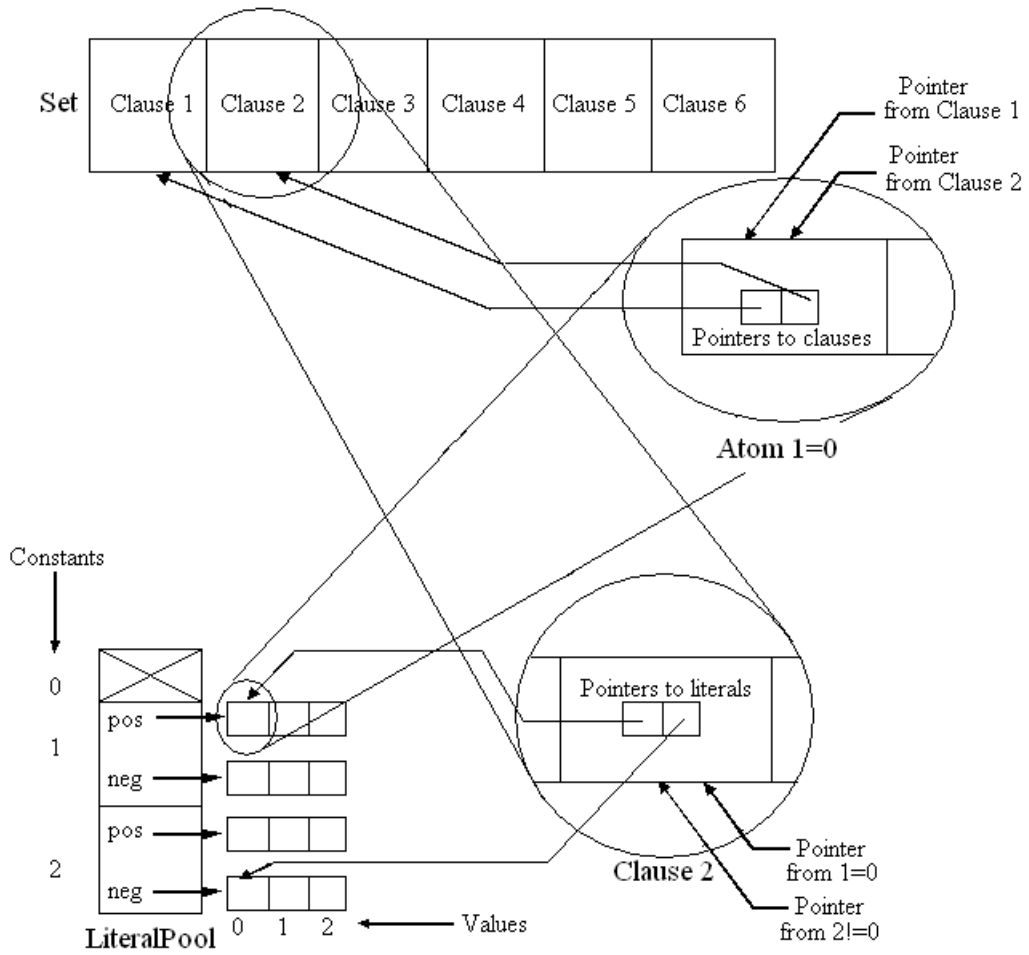


Figure 18: Example of Data Structures

Clause 2:  $\{1=0, 2 \neq 0\}$

Clause 3:  $\{1 \neq 1, 2=1\}$

Clause 4:  $\{1 \neq 2, 2 \neq 2\}$

Clause 5:  $\{1=1, 2=0\}$

Clause 6:  $\{1 \neq 0, 2=2\}$

## **4.3 Functionalities**

Let's now look at the seven functionalities listed earlier. Most of them are implemented as columns, meaning a highest level class provides the interface and in most cases does minimum amount of work before delegating the responsibility to a lower level class. Thus the functionalities are broken and spread over pillars or columns of classes. This way object oriented design is maintained and classes know only as much as they need to know.

### **4.3.1 Empty theory detection**

The class Set maintains a data member that keeps track of the number of unsatisfied clauses in the theory. So when this number reaches 0, we know that the theory is empty and the problem is solved. So all of the functionality is provided by the Set class.

### **4.3.2 Empty clause detection**

This is a column fashioned functionality spread over classes Set and Clause. The function in the Clause class checks the private variable of the class that keeps track of the number of literals in the clause that still have default status. If this variable has reached 0, the clause is empty and the function returns true.

The function in the Set class checks its Clause objects one by one for being empty and returns true the moment it detects the first empty clause. If no clause is found empty, it eventually returns false.

### **4.3.3 Unit clause detection**

This functionality is just like empty clause detection. The only difference is in the check that is performed in the Clause class. Instead of checking if the variable (number of default literals) is 0, it checks to see it is 1. If it is then the clause has only one default literal and so the clause is a unit clause. Therefore the function returns true. Function in the Set class remains unchanged.

#### 4.3.4 Entailment handling

At the highest level, the functionality is supported by LP. But LP only does the job of calling the corresponding function of all the LPN objects it has. Every LPN object then checks the following set of conditions using a low level accessor function of its negative literals. Here  $c$  is assumed to be the constant corresponding to this LPN object.

**Conditions:** For some  $v \in \text{dom}(c)$  we have,

- (i)  $c=v$  and  $c\neq v$  are both unassigned.
- (ii) All the  $c\neq v'$  ( $v' \in \text{dom}(c)$  and  $v' \neq v$ ) are satisfied.

If these conditions are met,  $c=v$  is entailed (and hence returned by the function).

#### 4.3.5 Pure literal detection

To detect pure literals efficiently LPN maintains two variables called *totPosSatCount* (total Positive Satisfaction Count) and *totNegSatCount* (total Negative Satisfaction Count). If  $c$  is the constant that this LPN represents then

$$\begin{aligned} \text{totPosSatCount} &= \sum_{v \in \text{dom}(c)} (\text{Number of unsatisfied clauses containing } (c=v)) \\ \text{totNegSatCount} &= \sum_{v \in \text{dom}(c)} (\text{Number of unsatisfied clauses containing } (c\neq v)) \end{aligned}$$

To detect positive pure literal (pure atom), LPN retrieves the number of unsatisfied clauses from all its positive literals (atoms) one by one. Let  $c=v$  be the current atom and the value retrieved be  $n$  i.e.  $c=v$  occurs in  $n$  unsatisfied clauses. There are three possible cases on  $n$ .

- (1) If  $n = \text{totPosSatCount}$ , it means no other  $c=v'$  ( $v \neq v'$ ) occurs in the theory. So  $c=v$  can be pure if  $c\neq v$  does **not** occur in the theory. LPN, therefore, checks to see if  $c\neq v$  is present in the theory. If it is not, then  $c=v$  is returned as a pure atom. Otherwise, no atom on  $c$  can be pure (because both  $c=v$  and  $c\neq v$  occur in the theory). Hence the search for a pure atom on this constant terminates immediately.

- (2) If  $n=0$ , then although  $c=v$  is not pure, there can still be some  $v' \in \text{dom}(c)$  ( $v' \neq v$ ), such that  $c=v'$  is pure. So the search continues with subsequent values  $v'$  of  $c$ .
- (3) If  $0 < n < \text{totPosSatCount}$ , no atom on  $c$  can be pure. So the search for pure atom on  $c$  terminates immediately.

In most of the cases,  $0 < n < \text{totPosSatCount}$ , so the search stops immediately, avoiding useless expensive subsequent checks.

When search for a pure atom fails, that for negative pure literal begins. Detecting a negative pure literal is even easier. Here all we have to check for a negative literal  $c \neq v$  is

- that  $c \neq v$  is unassigned and does occur in the theory and
- that  $c=v$  does not occur in the theory.

To check these conditions there is a function defined in the Literal class that returns the number of unsatisfied clauses **only if the literal is unassigned**. Otherwise it returns -1. The function is called *getNumOfClausesIfUnassigned*. LPN invokes *getNumOfClausesIfUnassigned* on all its negative literals and returns the first literal whose return value is greater than 0. If no such literal is found, NULL is returned indicating absence of negative pure literal.

#### 4.3.6 Branching: Computation of heuristic

As mentioned earlier, to pick an atom to branch on we tried five different heuristics.

**Heuristic 1: Highest Satisfaction Under Maximum Impact** This heuristic first picks a constant  $c$  that will maximize the impact and then picks a value  $v$  for  $c$ , such that  $c=v$  will satisfy maximum number of clauses. Impact of every constant is computed at its corresponding LPN object, by adding *totPosSatCount* and *totNegSatCount*. The function in LPN that does this is called *getImpact*. LP calls *getImpact* on all its LPN objects and picks the constant  $c$  whose LPN object returned the highest impact value. LP then delegates

the responsibility of picking a value  $v$  for  $c$  to the chosen LPN, by calling its *pickAtom* function. The function picks a value  $v$ , by retrieving the value of number of unsatisfied clauses from all its literals and using *totPosSatCount* and *totNegSatCount* stored in the LPN object.

**Heuristics 2, 3 and 4: Highest (*Satisfaction - Falsification*)** These functions are implemented by LP-LPN column. A function in LPN object compares the value of (*satisfaction - falsification*) for all the literals it is concerned with and returns the winner to LP. In LP, winners from all the LPN objects are compared and the final winner is picked.

**Heuristic 5: Highest *Satisfaction*** Implementation of this function is just like Heuristics 2, 3 and 4. This is also supported by LP-LPN column. A function in LPN object compares the value of *satisfaction* for all its atoms and returns the winner atom to LP object. LP object compares winners from all the LPN objects and picks the final winner.

#### 4.3.7 Backtracking: Avoiding multiple copies of the theory

As we advance the search we have to maintain the ability to backtrack to any earlier point higher up in the search tree. Whenever a literal is satisfied, some clauses get satisfied, some literals get “removed”. These elements (satisfied clauses and removed literals) are to be treated as if they don’t exist in the theory, since they do not contribute to the part of the interpretation that is constructed subsequently. Thus at every point in the search tree, there is a unique snapshot of the theory that we have to remember and that we may have to backtrack to. The easiest way is to make copies of the theory and associate a copy with a point in the tree. But this would be greatly uneconomical. To solve the problem more efficiently, we assign a level to every choice point in the search tree beginning with 0 (i.e. root at level 0). We maintain a global level counter that is incremented at every choice point. All the literals and clauses have a local variable to keep track of the level at which they change the status (i.e. get satisfied or falsified). Initially this variable is “unassigned” for all the elements. The level does not have any meaning unless the element is satisfied (for clause) or satisfied/falsified

(for literal). Whenever a literal becomes satisfied or falsified, we store with it the current value of the global level counter. Similarly whenever a clause becomes satisfied, we store the current value of the global level counter with it also. Thus all the clauses and literals that change status between two successive choice points in the tree have the same level. When it comes to backtracking to a level say  $l$ , we “undo” all the assignments whose level is greater than or equal to  $l$ . More specifically, default status is given to:

- all the satisfied clauses with level  $\geq l$ , and
- all the satisfied/falsified literals with level  $\geq l$ .

Thus the ability to backtrack is achieved without making multiple copies of the theory.

## 5 Evaluation Procedure and Experimental Results

Finite domain problems are not yet standardized, so there are no standard benchmark problems. Therefore, to run the experiments and compare the performance of our solver with others, we wrote our own generator to create random satisfiable instances of finite domain problems. We allow the user to specify the following parameters:

- (1) size of a clause (single value or a range),
- (2) domain size of a literal (single value or a range),
- (3) number of clauses per theory (single value or a range),
- (4) number of constants in a theory.

In this chapter we talk about the kind of instances generated and the comparisons made.

### 5.1 Syntax

We use an extended version of DIMACS-CNF format [5]. This is the standard input format for Boolean SAT solvers. Figure 19 shows our extension of DIMACS-CNF format. It has two sections: preamble and body. The preamble starts with optional comment lines. There can be any number of comment lines, but each must begin with `c`. For example, following line is a comment line.

```
c This is a comment line.
```

Comment lines are immediately followed by a problem line. A problem line looks like this:

```
p cnf <num of const> <num of clauses>
```

`<num of const>` and `<num of clauses>` are to be replaced with the actual number of constants and the number of clauses in the theory, respectively.

Next we have domain lines in the preamble. A domain line looks like this:

```

c This is a sample cnf file.
c Lines beginning with c are comment lines
c and are ignored.
c There can be any number comment lines
p cnf 3 4
d 1 3
d 2 4
2=1 3!=1 1!=2 0
1=1 2!=3 0
3!=0 2=0 0
1=0 3=1 2!=2 0

```

Figure 19: Sample CNF file

```
d <const> <domain size>
```

The `d` indicates that this is a domain line. `<const>` is replaced by a constant and `<domain size>` by its domain size. The line says that the constant `<const>` has domain  $\{0,1,2,\dots,<domain size>-1\}$ .

We can have at most one domain line for each constant in the theory. The constants that do not have any domain line are considered Boolean by default and so have domain  $\{0,1\}$ . Domain lines are not present in DIMACS-CNF format because all the constants are Boolean. The default mechanism of adopting Boolean domains contributes to the ability of our solver to run on DIMACS-CNF format. Preamble ends with the domain lines.

The body has `<num of clauses>` clauses. Here is a sample clause: `2=1 1!=3 5=7 0`. We used `!=` to represent  $\neq$ . The clause has three literals: `2=1`, `1!=3` and `5=7`. `0` is the delimiter.

Let's look at the literal  $1 \neq 3$  more closely. Here the constant is 1, value is 3 and relation is "not equal to". Whenever constant 1 is assigned a value other than 3 this literal is satisfied. Similarly, in case of  $2=1$ , 2 is the constant, 1 is the value and "equal to" is the relation. This literal gets satisfied only when the constant 2 is assigned value 1. The values that occur in literals must be in the domains declared (implicitly or explicitly) earlier in the preamble.

Finally, to make the solver compatible with the standard DIMACS-CNF format only one more adjustment is needed. Whenever the solver encounters a constant  $c$  or its negation, written as  $\neg c$ , (instead of  $c=v$  or  $c \neq v$ ) these are treated as  $c=1$  and  $c=0$  respectively.

## 5.2 Evaluation Procedure

We generated satisfiable problem instances with clause size 3 and domain size 5. In all the experiments described in this chapter these parameters are fixed unless mentioned otherwise. Also all the theories contain negative literals unless mentioned otherwise. In all the experiments below  $r$  stands for the ratio  $\frac{\text{number of clauses}}{\text{number of constants}}$ . We performed four sets of experiments:

- comparison of the five **heuristics**,
- comparison on theories **with** and **without** negative literals,
- comparison with **Sinha's** solver, and
- comparison with **zChaff**.

Now let's look at each of the sets in detail.

### 5.2.1 Heuristics Comparison

In this comparison we used two criteria – number of backtracks and search time. From our preliminary experiments we observed that Heuristic 3 performed considerably worse than all others, in every single case on both the criteria. Also Heuristic 2 and Heuristic 4 always performed almost the same. So we decided to

Number of Clauses	Heuristic 1	Heuristic 4	Heuristic 5
1000	0.1325096	0.24453	0.2438619
5000	3.888994	7.312347	7.265771
10000	19.40584	35.75721	35.67959
15000	50.8891	86.16848	86.09395

Table 1: Search Time in Seconds for  $r = 5$

drop Heuristic 2 and Heuristic 3 from further consideration. One of the reasons for choosing Heuristic 4 over Heuristic 2 is that Heuristic 4 employs the same criterion as Heuristic 2 but does less work than Heuristic 2 by not considering negative literals.

We generated theories with 1000, 5000, 10000 and 15000 clauses and kept the ratio  $r$  fixed at 5 and 10. In each of the four categories of  $r = 5$ , we generated 10 random satisfiable instances and observed that the number of backtracks was 0 for all the three heuristics in all the runs. Hence we display only search time for  $r = 5$ . Table 1 shows the search time in seconds required with each heuristic for  $r = 5$  and Figure 20 shows the corresponding graph. Thus we see that Heuristic 1 performed better than the other two here.

In the next set of experiments we kept  $r = 10$  and again generated theories with 1000, 5000, 10000 and 15000 clauses. Here in our preliminary experiments we observed that Heuristic 1 and Heuristic 5 caused a number of time outs when we experimented with only 10 instances in each category. The time out value was 100 sec here. So we generated 25 instances in each category and increased time out value to 300 sec. Of these 25, we tabulate here the averages of the ones that did not time out and provide the number that timed out in each category. For instance, an entry “15.2857 (3/10)” says 15.2857 is the mean number of backtracks in that category based on 7 instances, because 3 out of 10 instances timed out. This information is provided in table 2 along with the average number of backtracks. Table 3 shows the corresponding search time averages in seconds. Figures 21 and 22 show the graphs for the number of backtracks and search time respectively.

In these experiments we observed that on 15000 clauses only Heuristic 4 could perform without timing

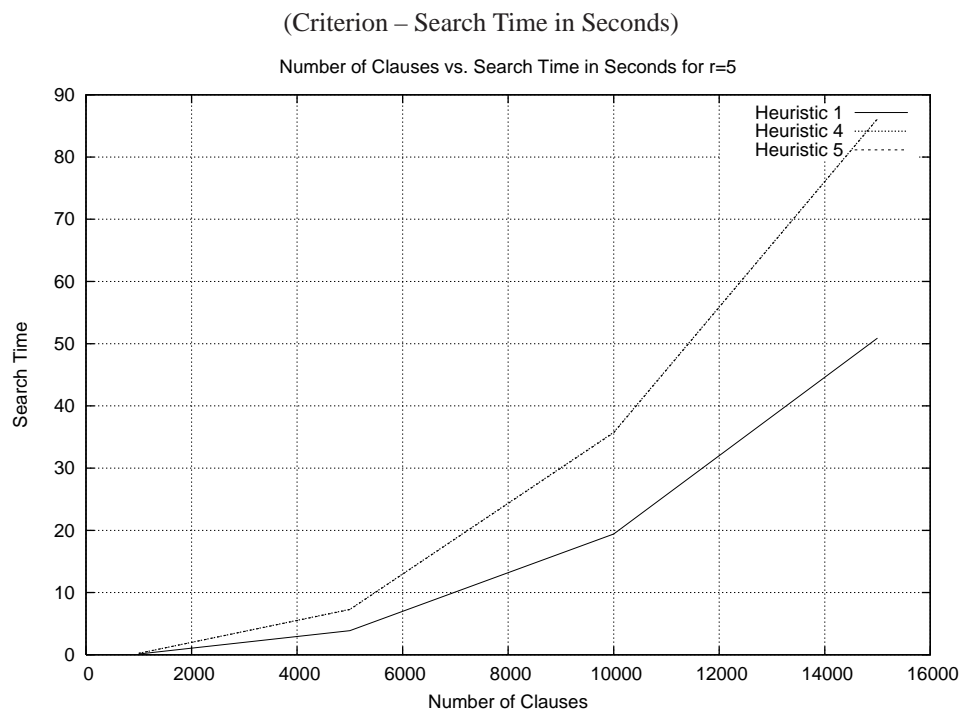


Figure 20: Heuristics Comparison for  $r = 5$

Number of Clauses	Heuristic 1 (T.O./Total)	Heuristic 4 (T.O./Total)	Heuristic 5 (T.O./Total)
1000	0.5 (0/10)	1 (0/10)	2.5 (0/10)
5000	15.2857 (3/10)	0.1111 (1/10)	2.5 (1/10)
10000	31.4444 (1/10)	0 (2/10)	0.875 (2/10)
15000	57.5384 (11/25)	2.4 (0/25)	115.2632 (6/25)

T.O. – Number of instances timed out

Total – Total number of instances

Table 2: Number of Backtracks for  $r = 10$

out even once. This is followed by Heuristic 5 at 6 time outs and finally Heuristic 1 with 11 time outs. Also the number of backtracks for Heuristic 4 is very low, as can be seen from Table 2.

Thus we see that as the problem size grows all the heuristics take longer. Some even start timing out. Their number of backtracks also grows. But Heuristic 4 not only keeps from timing out but also does not cause too many backtracks. We therefore conclude that Heuristic 4 is the overall winner. Between Heuristic 5 and Heuristic 1, we feel that Heuristic 5 performs better, although its number of backtracks grows with problem size, because the number of time outs caused by it is less and also its average search time is better. It is interesting though that Heuristic 1 is best on  $r = 5$  problems, which are clearly easier to solve. We speculate that this is because Heuristic 1 is easier to compute than Heuristic 4 and Heuristic 5. Heuristics 4 and 5 compare all the atoms in the theory, whereas Heuristic 1 considers only the ones that involve the constant with highest impact. Secondly, since for these easy problems there are many satisfying interpretations, search never backtracks. So the only factor that decides the speed is how fast a Heuristic finds a solution and not how many times it has to backtrack in doing so.

**Further Experiments on Heuristic 4:** Because of the performance of Heuristic 4, we decided to explore the family of heuristics using  $(\alpha \times \text{satisfaction} + \beta \times \text{falsification})$  further. We implemented two more heuristics, one with  $\alpha = 2$  and  $\beta = +1$  and the other with  $\alpha = 2$  and  $\beta = -1$ . Thus we implemented heuristics that use  $(2 \times \text{satisfaction} + \text{falsification})$  and  $(2 \times \text{satisfaction} - \text{falsification})$  as their

Number of Clauses	Heuristic 1	Heuristic 4	Heuristic 5
1000	0.0506	0.1084	0.1129
5000	2.237	2.7586	2.8007
10000	10.8479	13.4013	13.0773
15000	30.2906	19.1961	35.4882

Table 3: Search Time in Seconds for  $r = 10$

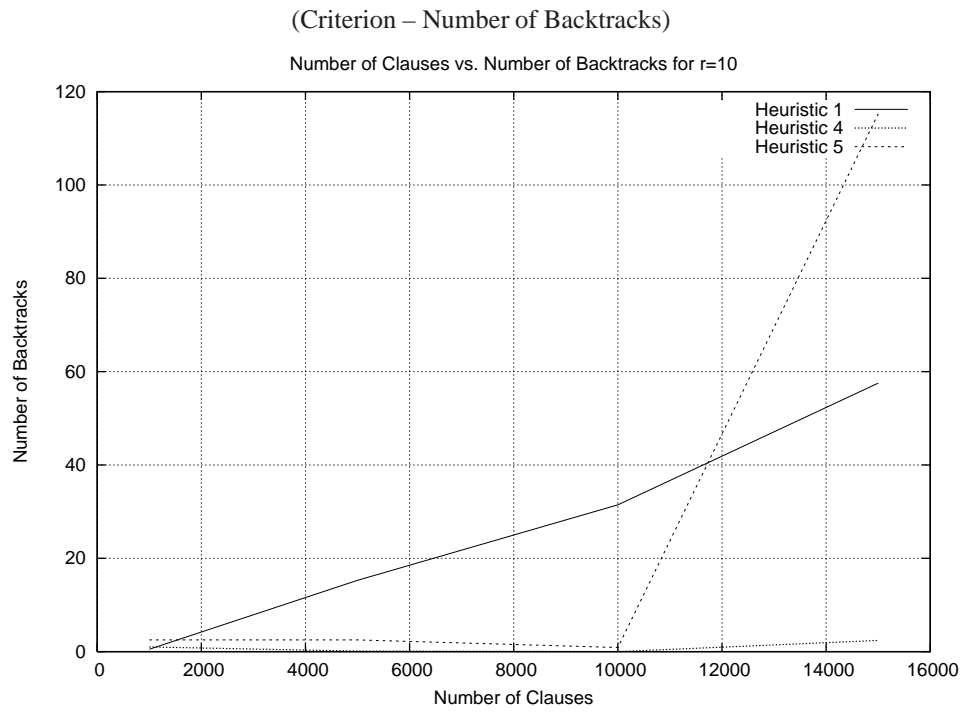


Figure 21: Heuristics Comparison for  $r = 10$

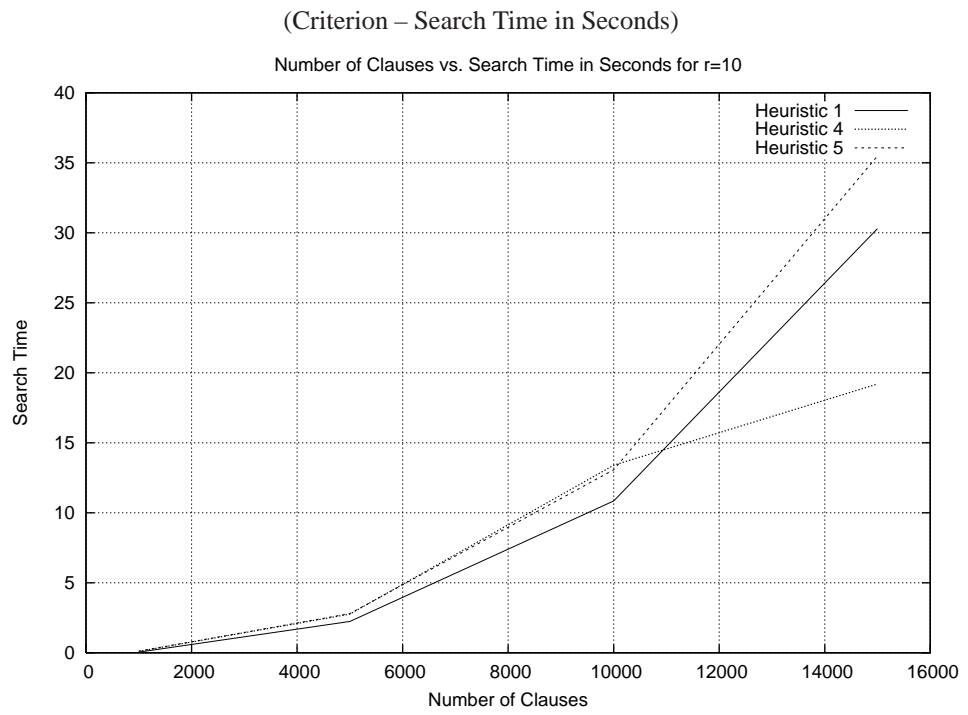


Figure 22: Heuristics Comparison for  $r = 10$

Number of Clauses	Heuristic 4 (T.O./Total)	2S+F (T.O./Total)	2S-F (T.O./Total)
1000	1 (0/10)	0.5556 (1/10)	0.75 (2/10)
5000	0.1111 (1/10)	19 (3/10)	5.3 (0/10)
10000	0 (2/10)	30.1429 (3/10)	12.4 (0/10)
15000	2.4 (0/25)	34.5454 (14/25)	14.9091 (3/25)

T.O. – Number of instances timed out

Total – Total number of instances

Table 4: Number of Backtracks for  $r = 10$

Number of Clauses	Heuristic 4 (T.O./Total)	2S+F (T.O./Total)	2S-F (T.O./Total)
1000	0.1084	0.0551	0.0559
5000	2.7586	2.2917	1.8695
10000	13.4013	9.3138	7.9037
15000	19.1961	46.6364	36.5007

Table 5: Search Time in Seconds for  $r = 10$

criterion and compared their performance with Heuristic 4. We ran them on the same set of problems as described earlier and observed that for  $r = 5$  the search times were increasing and the number of backtracks zero. Here we report the results for  $r = 10$  only in Table 4 and Table 5. As before, we also display the total number of instances and the number timed out in each case only in Table 4 but they apply to both the tables. Figure 23 and Figure 24 show the graphs for the number of backtracks and search time, respectively.

Thus we observe that, neither of the new heuristics performed better than Heuristic 4. The one using  $(2 \times \text{satisfaction} - \text{falsification})$  performed better than the one using  $(2 \times \text{satisfaction} + \text{falsification})$ . The same fact is observed from the number of time outs also.

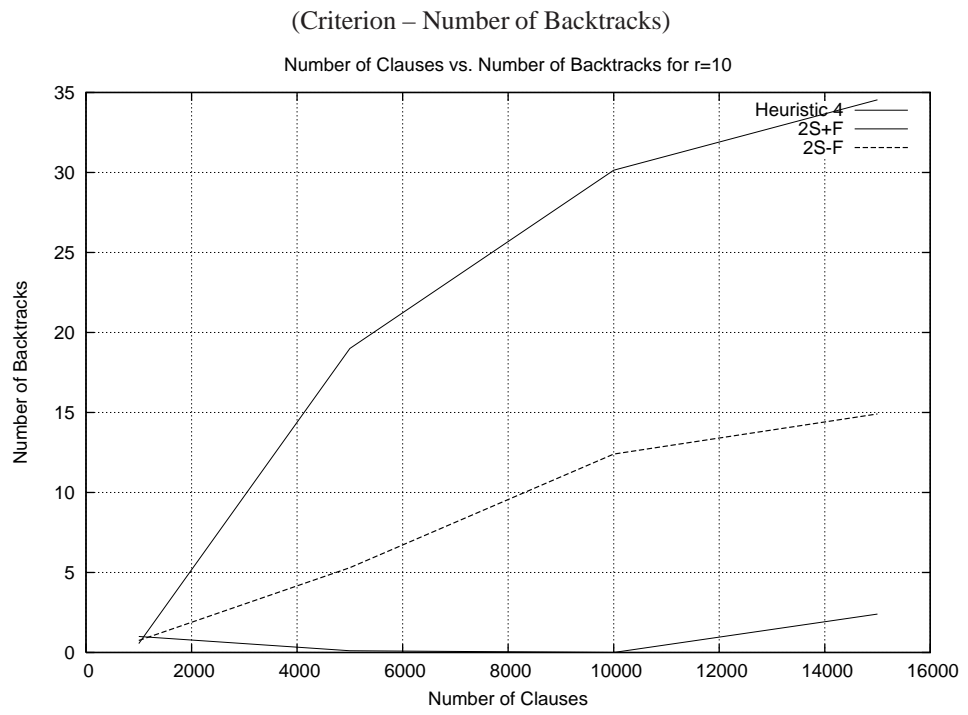


Figure 23: Comparison of Heuristic Family ( $\alpha \times \text{satisfaction} \pm \text{falsification}$ ) for r = 10

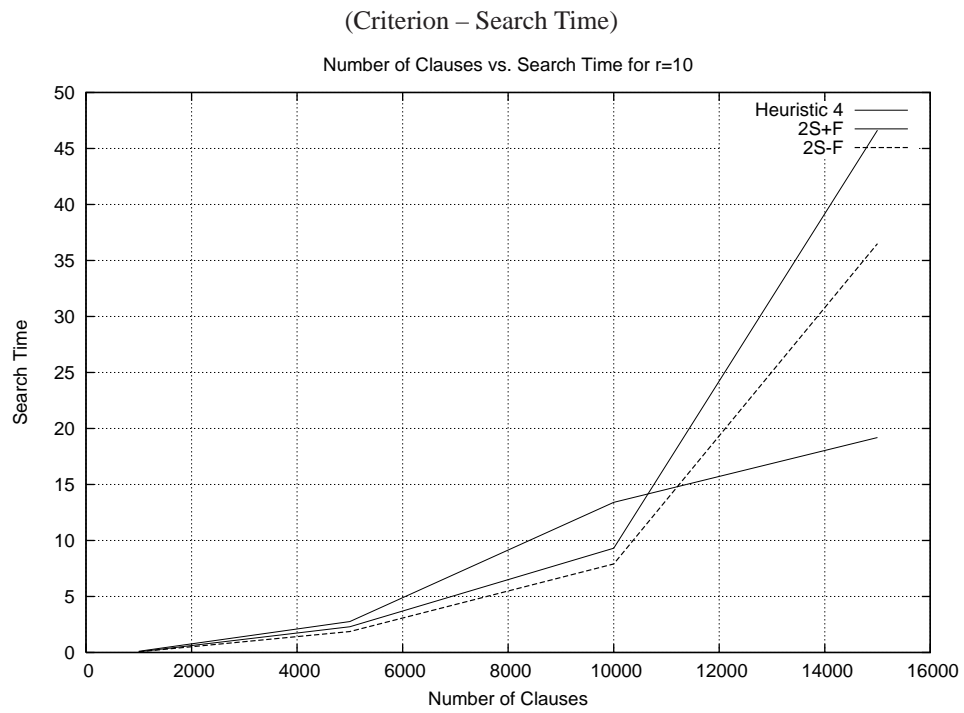


Figure 24: Comparison of Heuristic Family ( $\alpha \times \text{satisfaction} \pm \text{falsification}$ ) for  $r = 10$

Number of Clauses	$r = 5$		$r = 10$	
	With Negations (T.O)	Negations Compiled Out (T.O)	With Negations (T.O)	Negations Compiled Out (T.O)
1000	0.2445 (0)	0.1832 (0)	0.2836 (0)	6.0954 (0)
5000	7.3123 (0)	61.5534 (6)	12.4827 (1)	90.12 (9)
10000	35.7572 (0)	91.5886 (9)	30.7211 (2)	100 (10)
15000	86.1685 (0)	87.346 (8)	33.2802 (0)	100 (10)

T.O. – Number of instances timed out

Table 6: Search Time in Seconds for Heuristic 4

### 5.2.2 Comparison with and without Negative Literals Compiled Out

For this set of experiments we generated theories with 1000, 5000, 10000 and 15000 clauses and kept the ratio  $r$  at 5 and 10 for each of the categories. In all 8 categories we generated 10 random instances to average the values. Using the translation explained in chapter 2, we compiled the negative literals out from every theory and ran the solver on both kinds of datasets (i.e. the ones with negative literals in them and the ones with negative literals compiled out). We used two heuristics in this experiment – Heuristic 4 and Heuristic 5, since these performed better than Heuristic 1 in the previous set of experiments. In each run we recorded the search time. Table 6 and Table 7 show the average search time for Heuristic 4 and Heuristic 5, respectively. Figures 25, 26, 27 and 28 show the resulting graphs. Here also the averages reported are for the instances that did not time out. The time out value is 100 seconds. Figures in parentheses indicate the number of timed out instances in each case. The total number of instances is always the same, 10.

Thus we observe here that, except for one case (i.e. Heuristic 5,  $r = 5$ ), problems are solved much more efficiently when negative literals are retained in the theory.

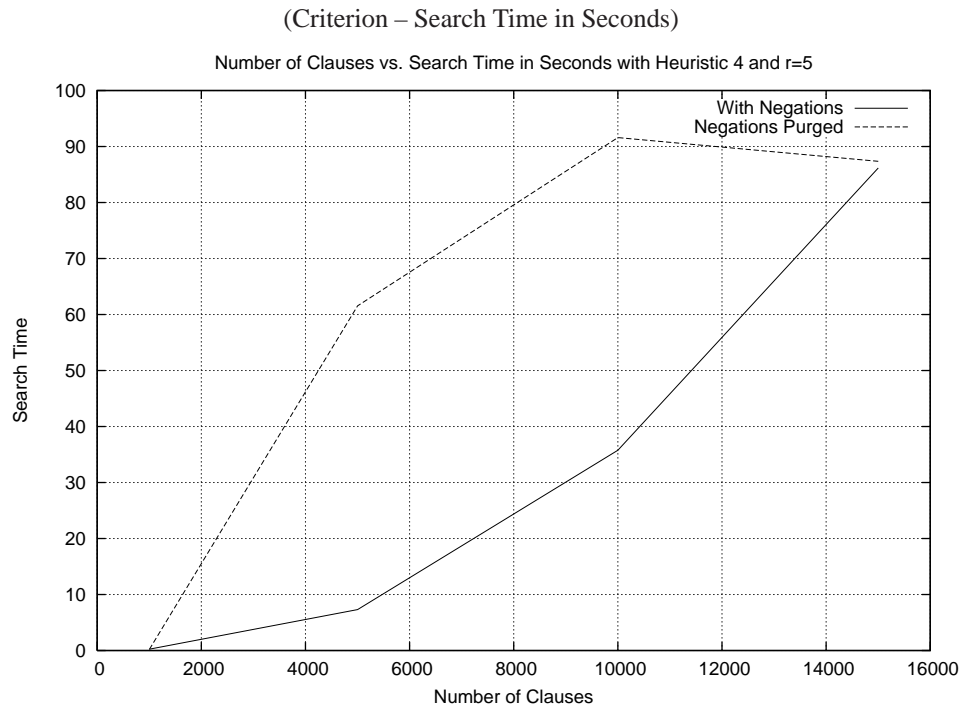


Figure 25: Comparison with and without Negative Literals with Heuristic 4 and r = 5

Number of Clauses	r = 5		r = 10	
	With Negations (T.O.)	Negations Compiled Out (T.O.)	With Negations (T.O.)	Negations Compiled Out (T.O.)
1000	0.2439 (0)	0.0487 (0)	0.1212 (0)	10.0159 (1)
5000	7.2658 (0)	12.6270 (1)	12.5973 (1)	80.0958 (8)
10000	35.6796 (0)	25.9631 (2)	30.4619 (2)	90.2070 (9)
15000	86.094 (0)	44.1035 (3)	62.4902 (4)	100 (10)

T.O. – Number of instances timed out

Table 7: Search Time in Seconds for Heuristic 5

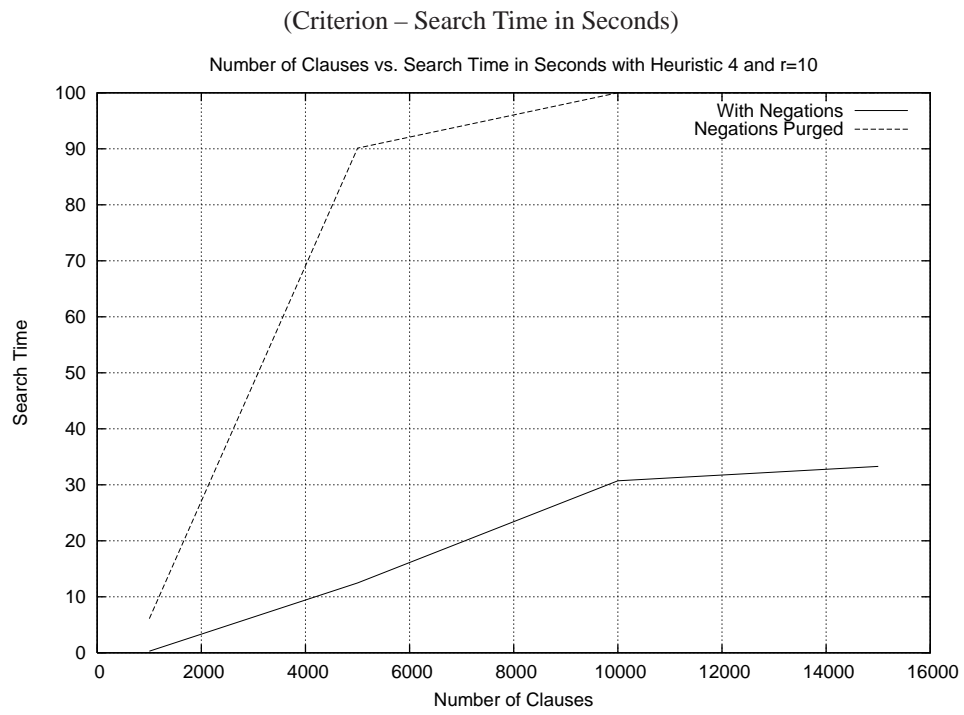


Figure 26: Comparison with and without Negative Literals with Heuristic 4 and  $r = 10$

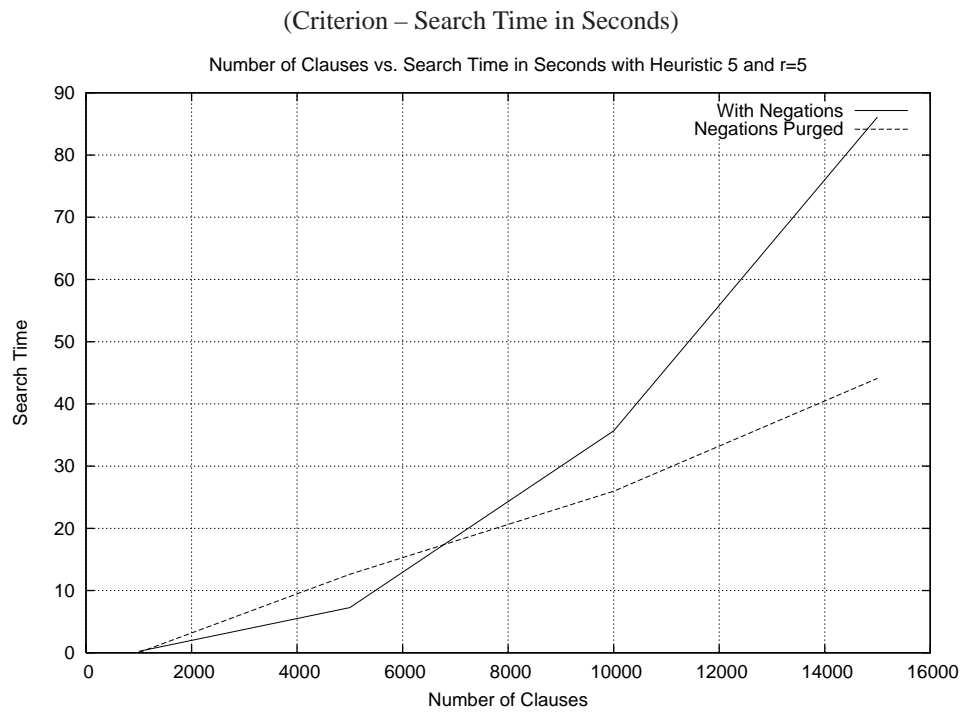


Figure 27: Comparison with and without Negative Literals with Heuristic 5 and  $r = 5$

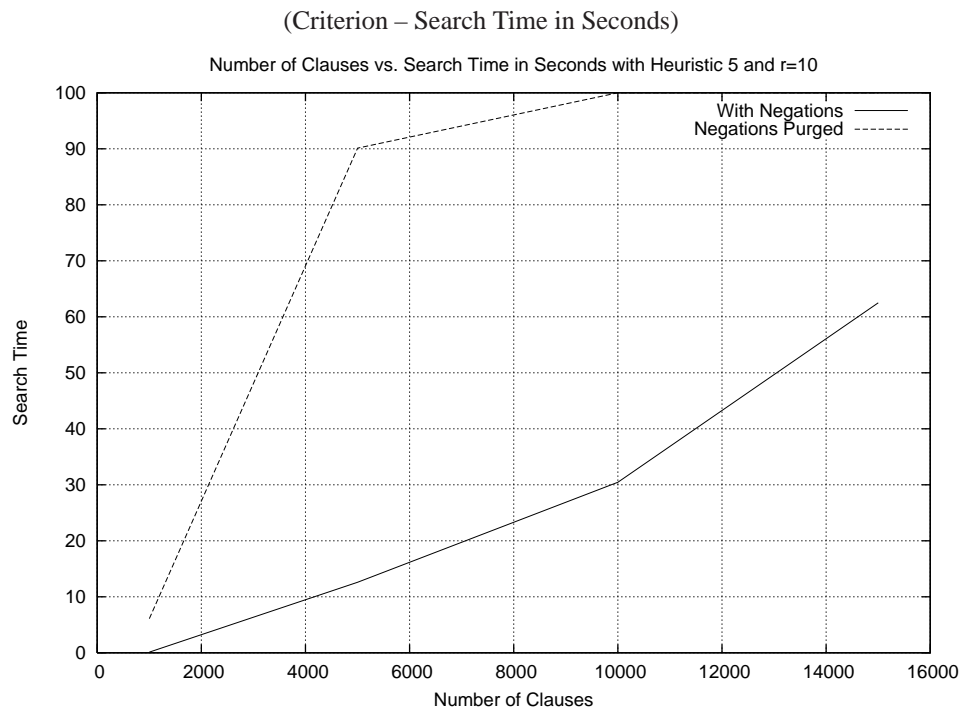


Figure 28: Comparison with and without Negative Literals with Heuristic 5 and  $r = 10$

Number of Clauses	$r = 5$		$r = 10$	
	Our Solver	Sinha's Solver	Our Solver	Sinha's Solver
1000	0.1275	0.104	0.0819	0.071
5000	1.894	2.013	0.8088	1.261
10000	3.8057	4.144	1.2322	2.343
15000	8.8267	12.7078	2.7699	6.654

Table 8: Execution Time in Seconds with Heuristic 5

### 5.2.3 Comparison with Sinha's Solver

In this set of experiments, our aim is to show that our solver performs as good as Sinha's solver on theories without negative literals, i.e. the theories that Sinha's solver is designed for. So we generated random theories (without negative literals) containing 1000, 5000, 10000 and 15000 clauses. We kept  $r$  fixed at 5 and 10. Thus with two values of  $r$  and four theory sizes we had eight categories, in each of which we generated 10 satisfiable random instances to average the results. We used execution time as the criterion of the comparison. In the runs of our solver we used Heuristic 5, because this is what Sinha uses in her solver. Table 8 shows the results. Figures 29 and 30 show the corresponding graphs for  $r = 5$  and  $r = 10$  respectively.

Thus we observe that in all the experiments our solver performed slightly better than Sinha's. Therefore even on the theories without negation our solver is at least as good as Sinha's solver and the improved performance due to allowing negation has not deteriorated the performance on the theories that do not have negation.

(Criterion – Execution Time in Seconds)

Number of Clauses vs. Execution Time in Seconds with Heuristic 5 and  $r=5$

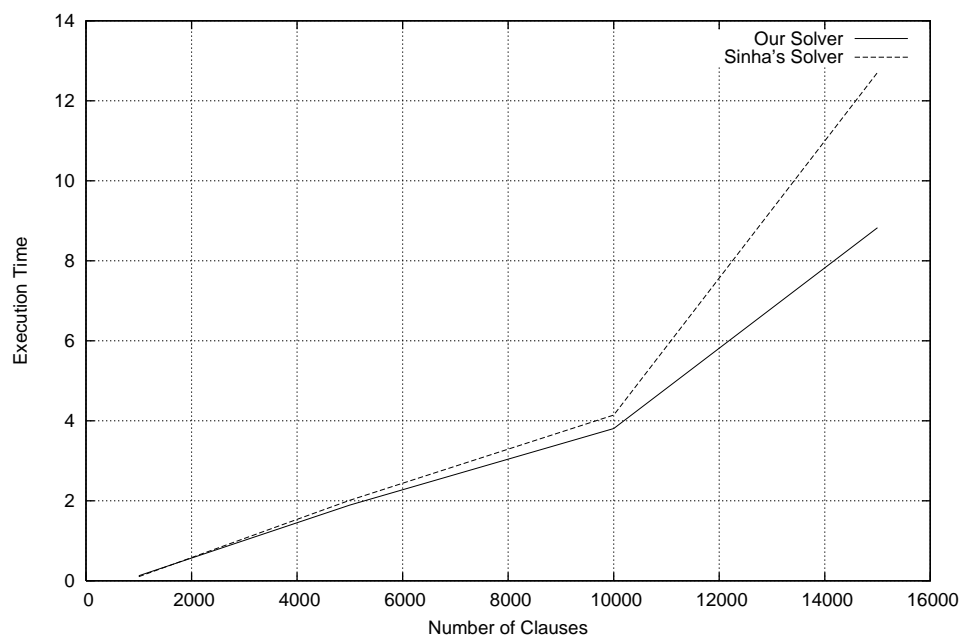


Figure 29: Comparison with Sinha's Solver with  $r=5$  and Heuristic 5

(Criterion – Execution Time in Seconds)

Number of Clauses vs. Execution Time in Seconds with Heuristic 5 and  $r=10$

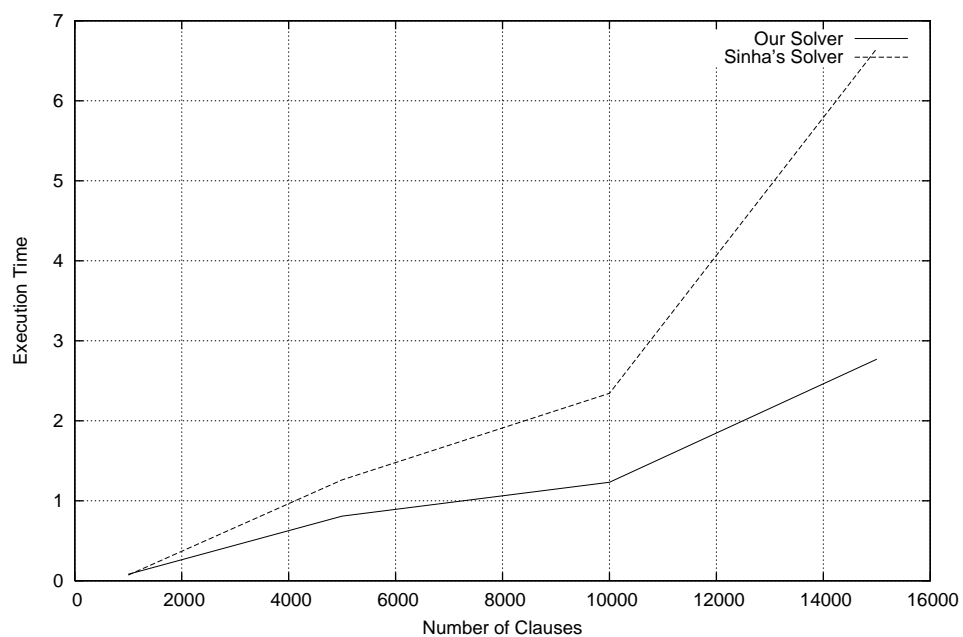


Figure 30: Comparison with Sinha's Solver with  $r=10$  and Heuristic 5

Number of Clauses	$r = 5$		$r = 10$	
	Our Solver	zChaff	Our Solver	zChaff
1000	0.279176	0.004	0.313307	0.002
5000	7.48669	0.027	12.6155	0.009
10000	36.11891	0.103	30.96223	0.023
15000	86.71717	0.167	33.74464	0.04

Table 9: Execution Time in Seconds with Heuristic 4

#### 5.2.4 Comparison with zChaff Solver

The aim of this set of experiments is to see how close we can get to the state-of-the-art Boolean solver, zChaff. We chose zChaff because Sinha has showed that other Boolean solvers like SATO can be outperformed with her (and so also our) approach. Sinha has also demonstrated that zChaff can be outperformed on quadratic translation. But zChaff was still beyond reach on linear translation. Therefore, we decided to try linear translation. We generated theories consisting of 1000, 5000, 10000 and 15000 clauses and kept  $r$  fixed at 5 and at 10. As before, we generated 10 instances in each category to average the values. zChaff reports total execution time, which is what is used as the basis of this comparison. Table 9 shows the results and Figures 31 and 32 show the corresponding graphs for  $r = 5$  and  $r = 10$ , respectively.

Although we were able to reach closer to zChaff than Sinha did, outperforming zChaff is still a challenge.

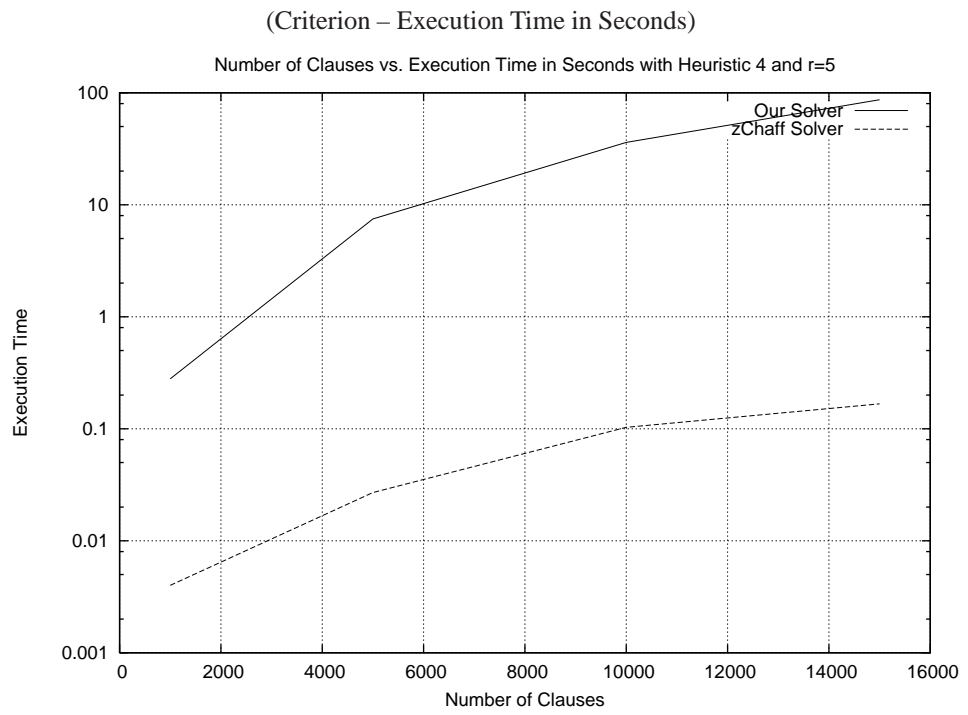


Figure 31: Comparison with zChaff Solver with r=5 and Heuristic 4

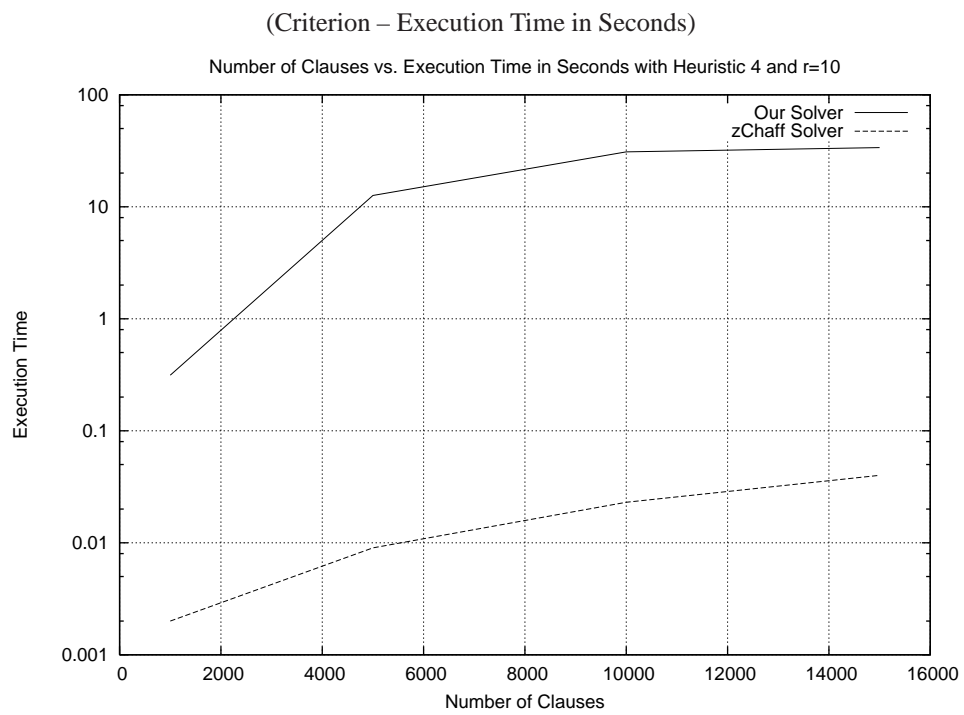


Figure 32: Comparison with zChaff Solver with r=10 and Heuristic 4

## 6 Conclusion

In this thesis we extended the finite domain DPLL algorithm introduced by Sinha [25] to handle negative literals. The experimental results demonstrated that we can take advantage of the negative literals present in the theory to improve efficiency. We also showed that our solver performs as well as Sinha’s solver on problems without negation – the class of problems that Sinha’s solver was designed for. Thus, the improved performance due to allowing negation does not cost us when no negation is present. On the other hand, our solver is still not competitive with the state-of-the-art Boolean solver, zChaff.

The comparison of heuristics verified that heuristics do affect the efficiency a lot. Overall Heuristic 4 performed better than others. Recall that Heuristic 4 is the one in which we pick the atom with highest (*satisfaction – falsification*). In our experiments we observed that as the problem size was increased, other heuristics started timing out but it was Heuristic 4 that could still solve all the problems in our largest problem size category (15000 clauses). The idea is that the heuristic picks such a literal that not only satisfies a high number of clauses (due to the term *satisfaction*) but will also keeps the number of reduced clauses low (due to the term *– falsification*) thereby making subsequent backtracks less likely.

### 6.1 Future Work

#### *Improvements in heuristics*

Zhang [17] discusses a number of effective heuristics for Boolean SAT solvers. One of the heuristics presented there, called Dynamic Largest Combined Sum (DLCS), is very similar to our Heuristic 1. They both use the same criterion (*satisfaction + falsification*). Therefore Heuristic 1 is just like the finite domain version of DLCS. We feel that it would be interesting to explore the use of  $(\alpha \times \textit{satisfaction} + \beta \times \textit{falsification})$  as the criterion in literal picking, experimenting with different values of  $\alpha$  and  $\beta$ . (Notice that Heuristics 4 and 5 are also members of this family.) Also there are a number of other heuristics suggested by Zhang that would be interesting to experiment with.

### *Improvements in the Data Structures*

Data structures is one of the most important factors affecting efficiency of the search. This is what we observed in our early attempts when we replaced the STL data structures in our implementation with custom ones. Even with our simple data structures we were able to see significant improvement in both speed and capability of the solver. Zhang [17] presents an account of more efficient data structures that can be used in the implementation of SAT solvers like ours. These can be adapted for finite domain DPLL.

### *More sophisticated backtracking*

The best SAT solvers use more sophisticated backtracking schemes than our solver does. Our solver implements “chronological backtracking”; when forced to backtrack it reverts to the most recent choice point at which the chosen literal is currently satisfied and then branches the other way by falsifying that literal. More sophisticated backtracking schemes are possible. Of particular interest are those that utilize some sort of analysis of the “reasons” for having obtained an empty clause in order to decide which choice point to back up to.

### *More random instances*

In most of the experiments we used 10 random satisfiable instances. If more instances are tried we might be able to address some of the partially answered questions, such as why Heuristic 5 performs better on  $r = 5$  problems, when negation is compiled out. It would also be interesting to see what happens on theories larger than 15000 clauses, and with ratios other than 5 and 10.

### *Evaluation on unsatisfiable instance*

In this thesis we only experimented with randomly generated satisfiable instances. It would be interesting to experiment with unsatisfiable instances also and see how quickly the solver can detect that the input theory is not satisfiable.

### *Solve real life problems*

One of the motivations of this thesis was that planning problems can be encoded nicely as finite domain problems. It will therefore be interesting to observe the performance of the finite domain solver on encodings of planning problems. We performed the experiments only on random instances, but experience with Boolean SAT solvers suggests that performance is likely to be rather different for real life problems.

### *Integration with CCALC*

CCALC [9] implements reasoning about actions and planning. Currently it makes use of Boolean SAT solvers, although, the input action description language of CCALC allows for finite domains for constants. CCALC can be modified to convert a planning problem into finite domain SAT problem (instead of Boolean SAT problem) and use a finite domain solver to solve it.

## References

- [1] A. M. Frisch and T. J. Peugniez: Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. In *Proc. of the Seventeenth Int'l Joint Conf. on Artificial Intelligence*, 2001.
- [2] C. Liu, A. Kuehlmann, and M. Moskewicz, CAMA: A multi-valued satisfiability solver. In *Proc. of IEEE/ACM Int'l Conf. on Computer Aided Design (ICCAD-03)*, San Jose, Nov. 2003.
- [3] C. Thiffault, F. Bacchus and T. Walsh: Solving non-clausal formulas with DPLL search. In *Proc. of Seventh Int'l Conf. on Theory and Application of Satisfiability Testing, SAT-2004*, 2004.
- [4] E. Giunchiglia, J. Lee, N. McCain, V. Lifschitz and H. Turner: Nonmonotonic causal theories. *Artificial Intelligence*, 2004.
- [5] <ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc>.
- [6] F. Bacchus: Constraints. *Unpublished manuscript available at <http://www.toronto.edu/fbacchus/constraints.pdf>*.
- [7] F. Bacchus: Enhancing Davis Putnam with extended binary clause reasoning. *National Conference on Artificial Intelligence (AAAI-2002)*, pages 613-619, 2002.
- [8] F. Bacchus and J. Winter: In *Selected Revised Papers Sat 2003 to be published by Springer-Verlag* , pages 341-355, 2003.
- [9] <http://www.cs.utexas.edu/users/tag/cc/>.
- [10] <http://www.itu.dk/Internet/sw2885.asp>.
- [11] <http://www.princeton.edu/chaff/zchaff/>
- [12] <http://www.satlive.org>

- [13] H. Herbstritt. zchaff: Modifications and Extentions. *Technical Report 188, Albert-Ludwigs-University, Germany*, 2003.
- [14] H. Zhang. SATO: An efficient propositional prover. In *Proc. of the Nat'l Conf. on Automated Deduction*, pages 272-275, 1997.
- [15] J. Marques Silva and K. A. Sakallah: Boolean satisfiability in Electronic Design Automation. In *ACM 2000*, 2000.
- [16] J. Marques Silva and K. A. Sakallah: Grasp: A new search algorithm for satisfiability. In *Proc. of the Nat'l Conf. on Computer Aided Design*, pages 220-227, 1996.
- [17] L. Zhang: SAT-Solving from Davis-Putnam and beyond. *Slides available at <http://research.microsoft.com/users/lintaoz/SATSolving/satsolving.htm>*
- [18] M. Davis, G. Logemann and D. Loveland: A machine program for theorem proving. *CACM 5*:394-397, 1962.
- [19] M. Davis and H. Putnam: A computing procedure for quantification theory. *Journal of ACM-7*, pages 201-205, 1960.
- [20] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. of the Design Automation Conference*, pages 530-535, 2001.
- [21] M. Vaziri, D. Jackson: Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. of Nineth Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS03)-Number 2619*, pages 505-520, 2003.
- [22] M. Velev, R. Bryant: Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation-35*, pages 73-106, 2003.

- [23] R. Bruni, A. Santori: Adding a new conflict based branching heuristic in two evolved DPLL SAT solvers. In *Proc. of Seventh Int'l Conf. on Theory and Application of Satisfiability Testing, SAT-2004*, 2004.
- [24] S. A. Cook: The complexity of theorem-proving procedures. In *Proc. of Third Annual ACM Symposium on Theory of Computing*, pages 151-152, 1971.
- [25] S. Sinha: A finite domain satisfiability solver. *Masters thesis available from the Computer Science Department of The University of Minnesota, Duluth*, 2003.
- [26] W. Kuchlin, C. Sinz: Proving consistency assertions for automotive product data management. *Journal of Automated Reasoning-24*, pages 145-163, 2000.