Watched Literals in a Finite Domain SAT Solver

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Anurag Jain

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

Dr. Hudson Turner

September 2008

UNIVERSITY OF MINNESOTA

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

ANURAG JAIN

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

Acknowledgements

I would like to take this opportunity to thank several people who have contributed directly and indirectly to the completion of my thesis work.

I begin by thanking my advisor Dr. Hudson Turner for providing me with an opportunity to work with him. I am grateful to Dr. Turner, whose invaluable guidence, encouragement and patience helped me complete this challenging yet interesting journey.

I would also like to thank my committee members, Dr. Doug Dunham and Dr. Pete Willemsen for their useful suggestions, and their time.

I would like to thank the faculty of the Department of Computer Science at UMD. Specifically, Dr. Hudson Turner, Dr. Pete Willemsen, Dr. Richard Maclin, and Dr. Ted Pederson for providing me an opportunity learn and excel in my career. I have a lot respect for the love and passion they have for what they do.

I would like to thank Dr. Carolyn Crouch for her support and encouragement.

I would like to thank the staff of the Department of Computer Science at UMD, especially, Jim Luttinen, Lori Lucia and Linda Meek for their support and encouragement during the past two years.

Finally, I would like to thank my parents for their selfless love. They continue to be the source of my inspiration and reason of my being.

Dedicated to my parents

Mrs. Kiran Jain and Mr. Rakesh Jain

and

my sister

Jyotsana Jain.

Abstract

Boolean Satisfiability (Boolean SAT) is a classic NP-complete problem and is used to represent many practical problems in mathematics, computer science and electrical engineering. Most solvers for Boolean SAT problems are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Over the last decade, there has been a remarkable improvement in the performance of DPLL-based Boolean SAT solvers due to innovations such as clause learning, non-chronological backtracking, watched literals and random restarts. In this thesis, we study solvers for a slight extension of the Boolean SAT problem, called Finite Domain SAT (FD SAT). Many practical applications can be more directly expressed using variables that can take values from a finite domain of a size greater than two (rather than the Boolean domain of $\{t, f\}$). To solve such FD SAT problems, it is possible to translate into Boolean format and solve using a Boolean SAT solver, but this approach increases the overall size of the problem and makes its representation somewhat more obscure. In this thesis, we develop and study an FD SAT solver that can be used to solve FD SAT problems directly, without converting them into Boolean format. This research is a continuation of work by Sinha, Nagle, and Lal. More specifically, in this thesis we study the trade-offs involved in moving from a "bookkeeping"-style FD SAT solver, as implemented by Lal, to an FD SAT solver that uses the watched literals method.

Contents

1 Introduction **1**

1.1 Finite Domain Satisfiability 2

1.2 Motivation and Related Work 3

1.3 Contribution of this Thesis 5

1.4 Outline of the Thesis 5

2 Finite Domain SAT **7**

2.1 Partial Interpretation and Algorithm 1 7

2.2 Unit Propagation and Algorithm 2 12

2.3 Algorithm 3 14

2.4 Clause Learning and Non-Chronological Backtracking and Algorithm 4 . . 21

2.5 Problem with Lal's FD-SAT Solver 28

3 Watched Literals and Algorithm 5 **31**

3.1 Comparisons and Conclusions 39

4 Heuristics **40**

4.1 Bookkeeping Heuristics and Algorithm 6 41

4.1.1 Bookkeeping Heuristic 1 44

4.1.2 Bookkeeping Heuristic 2 45

4.1.3 Bookkeeping Heuristic 3 45

4.2 Comparisons and Conclusions 46

5 More on Clause Learning and Non-Chronological Backtracking and Algorithm 7 47

6 Benchmark Problems and Experimental Results 60

6.1 *GT_N* 61

6.2 Pigeonhole 62

6.3 Pebbling 63

6.4 Random Problems 65

6.5 Comparison of FD-WL and FD-BK Solvers 66

6.5.1 Comparison on Backtrack Time 66

6.5.2 Comparison on Search Time, Number of Decisions, and Number of Backtracks 69

6.6 Heuristic Comparisons 71

6.6.1 Bookkeeping Heuristic 1 vs. VSIDS 72

6.6.2 Bookkeeping Heuristic 2 vs. VSIDS 77

6.6.3 Bookkeeping Heuristic 3 vs. VSIDS 81

7 Conclusion 86

7.1 Future Work 87

List of Tables

1. Comparison of backtrack time taken by FD-BK and FD-WL on Pigeonhole problems.
2. Comparison of backtrack time taken by FD-BK and FD-WL on Pebbling problems.
3. Comparison of backtrack time taken by FD-BK and FD-WL on GT_N problems.
4. Comparison of backtrack time taken by FD-BK and FD-WL on random satisfiable problems.
5. Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on Pigeonhole problems.
6. Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on Pebbling problems.
7. Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on GT_N problems.
8. Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on random satisfiable problems.
9. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pigeonhole problems.

10. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pebbling problems.

11. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on GT_N problems.

12. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on random satisfiable problems.

13. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pigeonhole problems.

14. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pebbling problems.

15. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on GT_N problems.

16. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on random satisfiable problems.

17. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pigeonhole problems.

18. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pebbling problems.

19. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on GT_N problems.

20. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on random satisfiable problems.

21. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pigeonhole problems.

22. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pebbling problems.

23. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on GT_N problems.

24. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on random satisfiable problems.

25. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pigeonhole problems.

26. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pebbling problems.

27. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on GT_N problems.

28. Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on random satisfiable problems.

29. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pigeonhole problems.

30. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pebbling problems.

31. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on GT_N problems.

32. Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on random satisfiable problems.

Chapter 1

Introduction

Boolean satisfiability (Boolean SAT) is a classic NP-complete problem. Boolean SAT is useful in expressing many practical problems in Artificial Intelligence, Group Theory, Formal Verification, Electronic Design Autmation (EDA), Logic Synthesis, etc. Problems in these fields expressed in Boolean SAT can be solved relatively efficiently using SAT solvers [2, 12, 18, 22]. Most algorithms used by Boolean SAT solvers are variations of the Davis, Putnam, Longemann, and Loveland (DPLL) algorithm [9, 10]. Recent developments such as clause learning, non-chronological backtracking, watched literals and random restarts have improved the performance of the Boolean SAT solvers tremendously [2, 3, 13, 17, 18, 19, 21, 22, 24].

Many problems discussed above can be nicely formulated as Finite Domain SAT (FD SAT) problems, which are an extension of Boolean SAT in which variables may have finite domains other than $\{t, f\}$. In general, FD SAT encodings retain the problem structure better than Boolean SAT encodings and thus it maybe useful to solve such problems directly using an FD SAT solver. In this thesis we build and study an FD SAT solver which extends the work done by Sinha [23], Nagle [20] and Lal [15] by incorporating a state-of-the-art technique called watched literals [18]. We do not attempt to optimize our watched literals implementation, but rather study the trade-offs in moving from a "bookkeeping" approach, as implemented by Lal, to the watched literals approach.

1.1 Finite Domain Satisfiability

We begin with a finite set V of symbols, called *variables*. Associated with each variable $v \in V$ is a non-empty finite set of symbols, called the *domain* of v, denoted by $dom(v)$. A *literal* is an expression of the form $v = x$ or $v \neq x$, where $v \in V$ and $x \in dom(v)$. A literal of the form $v = x$ is called *positive*; a *negative* literal on the other hand is of the form $v \neq x$. A *clause* is a finite set of literals. A *theory* is a finite set of clauses.

The question is, for a given theory, is there a mapping from variables to values from their domains that "satisfies" the theory? We make this precise as follows.

An *interpretation* is a function mapping each variable $v \in V$ to a value from $dom(v)$. An interpretation I *satisfies* a positive literal $v = x$ if $I(v) = x$, and *satisfies* a negative literal $v \neq x$ if $I(v) \neq x$. An interpretation *satisfies* a clause if it satisfies at least one of the literals from the clause and it *satisfies* a theory if it satisfies each of the clauses in the theory. An interpretation that satisfies a theory is known as a *model* of the theory. It is often convenient to represent an interpretation by the set of positive literals it satisfies.

The FD SAT problem then is to determine whether a given theory has a model. If there exists a model then the theory is *satisfiable*; otherwise it is *unsatisfiable*.

For example, take $V = \{A, B, C\}$, with $dom(A) = \{0, 1, 2\}$, $dom(B) = \{0, 1, 2, 3\}$, and $dom(C) = \{0, 1, 2, 3, 4\}$. Consider the following.

Satisfiable Theory:

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

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

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

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

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

Interpretation $\{A = 0,\ B = 2,\ C = 0\}$ satisfies the given theory and is therefore a model of the theory.

Unsatisfiable Theory:

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

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

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

The theory above is unsatisfiable.

1.2 Motivation and Related Work

Many standard Boolean SAT problems are expressed more naturally as FD SAT problems. There are a number of translations from FD SAT to Boolean SAT, such as Full Logarithmic Encoding [1], Full Regular Mapping [1], Linear Encoding [11], Quadratic Encoding [11], etc. These encoding schemes increase the size of the problem and may obscure its structure, thus making it harder to solve. This has led to our interest in FD SAT solvers, which can be directly used to solve FD SAT problems.

Let's look at the Finite Domain Encodings and Boolean Encoding of a Pigeonhole problem [18] involving fitting 4 pigeons in 3 holes.

Finite Domain Version $\{\{1 \neq 0,\ 2 \neq 0\}, \{1 \neq 0,\ 3 \neq 0\}, \{1 \neq 0,\ 4 \neq 0\}, \{2 \neq 0,\ 3 \neq 0\}, \{2 \neq 0,\ 4 \neq 0\}, \{3 \neq 0,\ 4 \neq 0\}, \{1 \neq 1,\ 2 \neq 1\}, \{1 \neq 1,\ 3 \neq 1\}, \{1 \neq 1,\ 4 \neq 1\}, \{2 \neq 1,\ 3 \neq 1\}, \{2 \neq 1,\ 4 \neq 1\}, \{3 \neq 1,\ 4 \neq 1\}, \{1 \neq 2,\ 2 \neq 2\}, \{1 \neq 2,\ 3 \neq 2\}, \{1 \neq 2,\ 4 \neq$

$2\}, \{2 \neq 2,\ 3 \neq 2\}, \{2 \neq 2,\ 4 \neq 2\}, \{3 \neq 2,\ 4 \neq 2\}\}$

The theory above consists of 4 variables each having domain of size 3. There are 18 clauses in the theory.

Boolean Version (Quadratic Encoding) $\{\{\neg 1 = 0,\ \neg 2 = 0\}, \{\neg 1 = 0,\ \neg 3 = 0\}, \{\neg 1 = 0,\ \neg 4 = 0\}, \{\neg 2 = 0,\ \neg 3 = 0\}, \{\neg 2 = 0,\ \neg 4 = 0\}, \{\neg 3 = 0,\ \neg 4 = 0\}, \{\neg 1 = 1,\ \neg 2 = 1\}, \{\neg 1 = 1,\ \neg 3 = 1\}, \{\neg 1 = 1,\ 4 = 1\}, \{\neg 2 = 1,\ \neg 3 = 1\}, \{\neg 2 = 1,\ \neg 4 = 1\}, \{\neg 3 = 1,\ \neg 4 = 1\}, \{\neg 1 = 2,\ \neg 2 = 2\}, \{\neg 1 = 2,\ \neg 3 = 2\}, \{\neg 1 = 2,\ \neg 4 = 2\}, \{\neg 2 = 2,\ \neg 3 = 2\}, \{\neg 2 = 2,\ \neg 4 = 2\}, \{\neg 3 = 2,\ \neg 4 = 2\}, \{\neg 1 = 0,\ \neg 1 = 1\}, \{\neg 1 = 1,\ \neg 1 = 2\}, \{\neg 1 = 0,\ \neg 1 = 2\}, \{\neg 2 = 0,\ \neg 2 = 1\}, \{\neg 2 = 1,\ \neg 2 = 2\}, \{\neg 2 = 0,\ \neg 2 = 2\}, \{\neg 3 = 0,\ \neg 3 = 1\}, \{\neg 3 = 1,\ \neg 3 = 2\}, \{\neg 3 = 0,\ \neg 3 = 2\}, \{\neg 4 = 0,\ \neg 4 = 1\}, \{\neg 4 = 1,\ \neg 4 = 2\}, \{\neg 4 = 0,\ \neg 4 = 2\}, \{1 = 0,\ 1 = 1,\ 1 = 2\}, \{2 = 0,\ 2 = 1,\ 2 = 2\}, \{3 = 0,\ 3 = 1,\ 3 = 2\}, \{4 = 0,\ 4 = 1,\ 4 = 2\}\}$

The theory above consists of 12 variables and 34 clauses. Each Boolean variable is of the form $x = y$, where $x = y$ is one of the positive literals from the finite domain version. The negation of $x = y$ is written $\neg x = y$. For example, $1 = 0$ is a Boolean variable and $\neg 1 = 0$ denotes its complement.

Sinha [23], Nagle [20], and Lal [15] provide evidence that solving FD SAT problems directly using a FD SAT solver may be advantageous. Similarly, Liu et al. [16] recently developed CAMA, which adapted speed-up techniques from state-of-the-art Boolean solvers such as clause learning, non-chronological backtracking, watched-literals and random restarts.

This thesis studies an FD SAT solver incorporating clause learning and non-chronological backtracking. Such a solver was implemented by Lal using a "bookkeeping" approach. In this thesis we attempt to study the trade-offs involved in switching from the "bookkeeping" approach to a state-of-the-art technique called "watched literals" [18].

1.3 Contribution of this Thesis

In this thesis we extend the work done by Sinha [23], Nagle [20] and Lal [15] on FD SAT solvers by incorporating the state-of-the-art watched literals technique in the solver. We do not attempt to optimize our watched literals implementation, but rather study the trade-offs in moving from a "bookkeeping" approach, as implemented by Lal, to the watched literals approach.

The FD SAT solver algorithm involves a search through the space of interpretations (or, rather, the space of ”partial interpretations”, as will be explained in Chapter 2). This search is usually guided by a ”heuristic” – a function that is used to try to estimate the potential usefulness of choices made in the course of the search. We compared the performance of several different heuristics.

In the course of this research, we discovered two serious problems with Lal’s FD SAT solver. The first is an algorithmic error which can cause Lal’s solver to crash on certain kinds of input. This is a rather subtle error that arises from a complication in adapting standard (Boolean) clause learning methods to the Finite Domain setting. The second error appears to be an implementation error, so in that sense it is less interesting. On the other hand, this error had the effect of producing remarkably good solution times for many of the difficult unsatisfiable benchmark problems. Thus, unfortunately, the most promising of Lal’s experimental results are invalid.

1.4 Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we develop a high level description of the FD DPLL algorithm using "bookkeeping". We also describe the implementation bug that invalidates the experimental results reported by Lal. Chapter 3 presents a "watched literal" version of the algorithm developed in Chapter 2. Chapter 4 presents a variation of the bookkeeping algorithm that is useful for experimenting with various heuristics. Chapter 5 presents information relevant for implementing clause learning and non-chronological backtracking, and describe the algorithmic error in the clause learning portion of Lal's FD SAT solver. In Chapter 6 we present the benchmark problem descriptions and experimental results. Finally, Chapter 7 concludes the thesis and suggests future work.

Chapter 2

Finite Domain SAT

This chapter begins with background knowledge concerning "partial interpretations" and "unit propagation", followed by elaborations of the DPLL algorithm leading to a high-level description of a version of the FD DPLL algorithm that incorporates clauses learning and non-chronological backtracking, using the "bookkeeping" approach.

2.1 Partial Interpretation and Algorithm 1

A *partial interpretation* is a set of literals such that at least one interpretation satisfies all literals in the set. If an interpretation I satisfies all literals in a partial interpretation P, we say I is an *extension* of P. Thus, a partial interpretation is a set of literals with at least one extension. Notice also that every interpretation (understood as the set of positive literals it satisfies) is also a partial interpretation (whose unique extension is itself).

A partial interpretation P *satisfies* a literal L, written $P \models L$, if every extension of P satisfies L. A partial interpretation P *satisfies* a clause C, written $P \models C$, if it satisfies at least one literal from the clause, and it *satisfies* a theory T, written $P \models T$, if it satisfies all the clauses in the theory.

Consider the theory $T = \{\{A = 0\}, \{A \neq 1, B = 2\}\}$, with $dom(A) = \{0,1,2\} = dom(B)$. Notice that partial interpretation P_1 = $\{A = 0\}$ satisfies literal $A = 0$, since every interpretation that satisfies $A = 0$ satisfies $A = 0$. Similarly, $P_1 \models A \neq 1$, since every interpretation that satisfies $A = 0$ satisfies $A \neq 1$. It follows that P_1 satisfies both clauses in T. Therefore, $P_1 \models T$. On the other hand, the partial interpretation $P_2 = \{A \neq 1\}$ does not satisfy T, because it does not satisfy the clause $\{A = 0\}$. Indeed, one of the (six) extensions of P_2 is the interpretation $I = \{A = 2, B = 1\}$, and since $I \not\models A = 0$, it follows that $P_2 \not\models A = 0$. So, $P_2 \not\models \{A = 0\}$ and thus $P_2 \not\models T$.

Notice that a theory is satisfiable if and only if some partial interpretation satisfies it. Indeed, if a theory T is satisfiable, then some interpretation I satisfies T, and as remarked above, I itself is a partial interpretation that satisfies T (since I is the unique extension of I). On the other hand, if a partial interpretation P satisfies T, then P has an extension I (by definition of partial interpretation), and I satisfies T (by definition of satisfaction) w.r.t a partial interpretation), so T is satisfiable.

So the FD SAT problem can be solved by determining whether a satisfying partial interpretation exists. More specifically, the space of partial interpretations can be searched using depth-first search to find a partial interpretation that satisfies the theory, if one exists. If all the partial interpretations have been searched and no satisfying partial interpretation is found then the theory is unsatisfiable.

In order to describe such a search precisely, it helps to define when a partial interpretation "falsifies" a theory, as follows.

A partial interpretation P *falsifies* a literal L, written $P \dashv L$, if it satisfies its complement, where the *complement* of a positive literal $v = x$ is the negative literal $v \neq x$, and the *complement* of a negative literal $v \neq x$ is the poitive literal $v = x$. We sometimes write $\bar{L}$ to denote the complement of literal L. Thus, $P \dashv L$ iff $P \models \bar{L}$. A partial interpretation *falsifies* a clause C, written $P \dashv C$, if it falsifies all the literals in C, and it *falsifies* a theory T, written

$P \models\!\!\!\!\dashv T$, if it falsifies at least one clause in T.

Algorithm 1

```
$T$ is a theory
**DPLL**($P$)
 if ($P \models T$) then
   return $P$
 if ($P \dashv T$) then
   return *false*
 $L \leftarrow$ pickLiteral($P$)
 if (DPLL($P \cup \{L\}$) $\neq$ *false*) then
   return DPLL($P \cup \{L\}$)
 return DPLL($P \cup \{\bar{L}\}$)
```

```
**pickLiteral**($P$)
 return a literal $L$ such that $P \not\dashv L$[1] and $L \in C$ for some $C \in T$ s.t. $P \not\models C$[2]
```

Given a theory T, the call **DPLL($\emptyset$)** will return a partial interpretation P such that $P \models T$, if one exists; otherwise *false* is returned.

Let's try to understand the algorithm with the help of a few examples.

Example 1

Consider a theory T with variables $V = \{1,2\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$. The theory consists of the two clauses shown below. Initially, we call **DPLL($\emptyset$)**.

$P = \emptyset$

$Clause1\ \{1 = 0\}$ $\quad U$

$Clause2\ \{1 = 1,\ 2 = 1\}$ U

The symbol U to the right of each clause indicates that the clause is "unassigned" with respect to the current partial interpretation P, i.e. P neither satisfies nor falsifies the clause. As the algorithm proceeds, the current partial interpretation P changes. We will write S to

[1] $A \not\dashv B$ represents A does not falsify B
[2] $A \not\models B$ represents A does not satisfy B

indicate that the current partial interpretation P satisfies the clause, and F to indicate that P falsifies the clause.

Since the current partial interpretation P neither satisfies nor falsifies the given theory T, we use **pickLiteral(P)** to obtain a literal L such that $P \not\models\!\!| L$ and L belongs to a clause in T that is not currently satisfied. (In general, the algorithm is nondeterministic at this point; any such literal will do.) By our choice of L, we know that $P \cup \{L\}$ is a partial interpretation. (Indeed, since P is a partial interpretation that does not falsify L, there must be at least one extension I of P such that $I \not\models \bar{L}$. It follows that $I \models L$. So I is an extension of $P \cup \{L\}$, which shows that $P \cup \{L\}$ is a partial interpretation.) Notice that we also know by choice of L that $P \cup \{\bar{L}\}$ is a partial interpretation. (Indeed, since L belongs to a clause that is not satisfied by P, it follows that $P \not\models L$. That is, there is an extension I of P s.t. $I \not\models L$. It follows that $I \models \bar{L}$, and so I is an extension of $P \cup \{\bar{L}\}$, which shows that $P \cup \{\bar{L}\}$ is a partial interpretation.)

Assume the chosen literal L is $1 = 0$ from $Clause1$. Now, we make the call **DPLL($\emptyset \cup \{1 = 0\}$)** and check whether the current partial interpretation $P = \{1 = 0\}$ satisfies or falsifies T.

$P = \{1 = 0\}$

$Clause1$ $\{$*1 = 0*$\}$ S

$Clause2$ $\{$~~$1 = 1$~~, $2 = 1\}$ U

As we see above, $Clause1$ is satisfied. Notice that literal 1=0 in $Clause1$ is italicized, indicating that literal 1=0 is satisfied by the current partial interpretation P. On the other hand, literal 1=1 in $Clause2$ has a strikethrough, indicating that literal 1=1 is falsified by P. But P neither satisfies nor falsifies 2=1, so $Clause2$ is still unassigned with respect to P. Thus, T is neither satisfied nor falsified with respect to P. So we will have to choose another literal.

So we next choose literal $2 = 1$ from $Clause2$, make the call **DPLL($\{1 = 0\} \cup \{2 = 1\}$),** and then check whether $P = \{1 = 0, 2 = 1\}$ satisfies T.

$P = \{1 = 0,\ 2 = 1\}$

$Clause1$ $\{$*1 = 0*$\}$ S

$Clause2$ $\{$~~1 = 1~~,*2 = 1*$\}$ S

As shown above, $Clause1$ and $Clause2$ are satisfied with respect to *P*, i.e. theory T is satisfied with respect to *P*. Since $P \models T$, partial interpretation *P* is returned, and we know that the theory is satisfiable, with model {1=0, 2=1} (which is the sole extension of *P*).

Example 2

Consider a theory with set of variables $V = \{1\}$ and with $dom(1) = \{0, 1\}$. The theory T consists of the two clauses given below. As usual, we begin by calling **DPLL(∅)**.

$P = \emptyset$

$Clause1$ $\{1 = 0\}\, U$

$Clause2$ $\{1 \neq 0\}\, U$

Since $P = \emptyset$ neither satisfies nor falsifies T, we pick a literal. Suppose the chosen literal is $1 = 0$ from $Clause1$. Now, we call **DPLL(**$\emptyset \cup \{1 = 0\}$**)** and check whether the current partial interpretation $P = \{1 = 0\}$ satisfies T.

$P = \{1 = 0\}$

$Clause1$ $\{$*1 = 0*$\}\, S$

$Clause2$ $\{$~~$1 \neq 0$~~$\}\, F$

As shown above, $Clause1$ is satisfied, but $Clause2$ is falsified with respect to *P*. So *T* is falsified with respect to *P*, and accordingly the call **DPLL(**$\{1 = 0\}$**)** returns *false*, so we "backtrack" and call **DPLL** with the complement of $1 = 0$, i.e. call **DPLL(**$\emptyset \cup \{1 \neq 0\}$**)**.

$P = \{1 \neq 0\}$

$Clause1$ $\{$~~$1 = 0$~~$\}\, F$

$Clause2$ $\{$*1 ≠ 0*$\}\, S$

Again *P* falsifies T, now because T falsifies $Clause1$. So **DPLL(**$\{1 \neq 0\}$**)** returns *false*.

Thus, **DPLL($\emptyset$)** returns *false*, signifying that T is unsatisfiable.

2.2 Unit Propagation and Algorithm 2

Notice that a model is simply an interpretation that satisfies at least one literal in each clause in the theory. Similarly, a partial interpretation P satisfies a theory T iff P satisfies at least one literal from each clause in T. This observation motivates the following. Let P be a partial interpretation. A clause C is *unit* with respect to P if $P \not\models C$ and P falsifies all but one literal L in C; L is called the *unit literal*.

In our depth-first search algorithm, if P is the current partial interpretation and P neither satisfies nor falsifies the theory T, then we are interested in whether some partial interpretation P' that is a superset of P satisfies T. If a clause C is unit w.r.t. P, then any such P' must satisfy the unit literal $L \in C$. So we can expedite our search by adding L to P. (Indeed, notice that $P \cup \{L\}$ is guaranteed to be a partial interpretation, since $P \not\models\!\!| L$. Moreover, since $L \in C$ and $P \not\models C$, we know that $P \not\models L$, so adding L to P advances our search for a superset of P that satisfies T.) In Algorithm 2 we incorporate this idea into the depth-first search algorithm.

Algorithm 2

```
T is a theory
DPLL(P)
 if ($P \models T$) then
  return P
 if ($P =\!\!| T$) then
  return false
 if there exists a unit literal L w.r.t. P then
  return DPLL($P \cup \{L\}$)
 L ←pickLiteral(P)
 if (DPLL($P \cup \{L\}$) $\neq$ false) then
  return DPLL($P \cup \{L\}$)
 return DPLL($P \cup \{\bar{L}\}$)
```

Let's look at an example of unit propagation. Consider a theory with variables $V = \{1,2\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$. The theory T consists of the four clauses given below. As always, we start with an empty partial interpretation $P = \emptyset$.

$P = \emptyset$

$Clause1$ $\{1 \neq 1\}$ U

$Clause2$ $\{1 \neq 2\}$ U

$Clause3$ $\{1 = 2,\ 2 = 2\}$ U

$Clause4$ $\{1 = 0,\ 2 = 1\}$ U

$Clause1$ is unit with respect to P, so $1 \neq 1$ is a unit literal. Accordingly, assume that we call **DPLL(**$\emptyset \cup \{1 \neq 1\}$**)**.

$P = \{1 \neq 1\}$

$Clause1$ $\{\mathit{1} \neq \mathit{1}\}$ S

$Clause2$ $\{1 \neq 2\}$ U

$Clause3$ $\{1 = 2,\ 2 = 2\}$ U

$Clause4$ $\{1 = 0,\ 2 = 1\}$ U

Similarly, $Clause2$ is unit with respect to P, and so $1 \neq 2$ is a unit literal. So we call **DPLL(**$\{1 \neq 1\} \cup \{1 \neq 2\}$**)**.

$P = \{1 \neq 1,\ 1 \neq 2\}$

$Clause1$ $\{\mathit{1} \neq \mathit{1}\}$ S

$Clause2$ $\{\mathit{1} \neq 2\}$ S

$Clause3$ $\{$~~$1 = 2$~~$,\ 2 = 2\}$ U

$Clause4$ $\{\mathit{1} = \mathit{0},\ 2 = 1\}$ S

Notice that the current partial interpretation $P = \{1 \neq 1, 1 \neq 2\}$ leaves only one possible value for variable 1, since every extension of P maps variable 1 to 0, which is the only remaining available value in $dom(1)$. Thus $P \models 1 = 0$, which is why P satisfies $Clause4$.

Now $Clause3$ is unit with respect to P, as $1 = 2$ is falsified by P. So we must satisfy unit literal $2 = 2$ in *Clause*3. Thus we call **DPLL(**$\{1 \neq 1, 1 \neq 2\} \cup \{2 = 2\}$**)**.

$P = \{1 \neq 1,\ 1 \neq 2,\ 2 = 2\}$

*Clause*1 $\{1 \neq 1\}$ S

*Clause*2 $\{1 \neq 2\}$ S

*Clause*3 $\{$~~$1 = 2$~~, $2 = 2\}$ S

*Clause*4 $\{1 = 0,$ ~~$2 = 1$~~$\}$ S

So **DPLL(**$\{1 \neq 1, 1 \neq 2, 2 = 2\}$**)** returns $\{1 \neq 1, 1 \neq 2, 2 = 2\}$, which is then returned by **DPLL(**$\{1 \neq 1, 1 \neq 2\}$**)** and so returned by **DPLL(**$\{1 \neq 1\}$**)** and so by **DPLL(**$\emptyset$**)**. This shows that $P = \{1 \neq 1, 1 \neq 2, 2 = 2\}$ satisfies T. And since $I = \{1 = 0, 2 = 2\}$ is an extension of P, we can conclude that I is a model of T.

2.3 Algorithm 3

In this section we will present a more detailed version of Algorithm 2 in which we begin to incorporate some standard ideas for how to implement the algorithm somewhat efficiently. This involves maintaining global data structures for representing the current partial interpretation and for keeping track of our progress toward satisfying the theory. Such structures can also simplify the task of detecting unit clauses.

Each literal L will have two associated fields: *value* and *level*. Field *L.value* represents whether L is satisfied, falsified or unassigned with respect to the partial interpretation P. If $P \models L$ then $L.value = t$. If $P \Dashv L$ then $L.value = f$. Otherwise $L.value = u$. Thus, the current partial interpretation is just the set $\{L \mid L.value = t\}$. *L.level* is a little harder to explain. If $L.value = u$, we do not care about *L.level*, but if $L.value = t$ or $L.value = f$, then *L.level* records the number of "decision literals" in P at the time that L was assigned its value, where a decision literal is a literal either obtained from **pickLiteral()** or obtained as the complement of such a literal (upon backtracking). (*L.level* is useful when backtracking

– it makes it easy to undo the appropriate assignments.)

So initially for each literal L, $L.value = u$, and $L.level$ is set to -1 (a dummy value).

There are similar fields associated with each clause C in the theory. $C.sat$ represents whether the clause C is satisfied or not with respect to P. If $P \models C$ then $C.sat = t$. If $P \not\models C$ then $C.sat = f$. If $C.sat = t$, $C.level$ represents the number of decision literals in P when C became satisfied. (Again, $C.level$ is useful when backtracking – we can easily determine when backtracking undoes the satisfying assignment for a clause.) $C.count$ is the number of literals L in C s.t. $L.value \neq f$. Notice that when $C.count = 1$ and $C.sat = f$, we know that C is unit w.r.t. the current partial interpretation. Similarly, when $C.count = 0$ we know that the clause, and so the theory, is currently falsified.

Now, let's have a look at the algorithm. Algorithm 3 has almost the same structure as Algorithm 2, with the addition of an initializing procedure, **Solve()**, which in turn makes the initial call **DPLL(0).** Here the argument 0 is the initial value of the parameter *level*, which is used to keep track of the current depth in the search space. That is, as in the previous discussion, *level* is the number of decision literals in the current partial interpretation.

Also, Algorithm 3 includes procedures **propagate()**, **backtrack()**, **checkSat()**, and **findUnitLiteral()** for using and maintaining the global data structures.

Algorithm 3

```
Solve()
 for all literals $L$ do
   $L.value \leftarrow u$
   $L.level \leftarrow -1$
 for all clauses $C$ do
   $C.sat \leftarrow f$
   $C.count \leftarrow$ number of literals in $C$
   $C.level \leftarrow -1$
 DPLL(0)
```

```
**DPLL**($level$)
  **if** (**checkSat**() = $t$) **then**
    **return** $\{L \mid L.value = t\}$
  **if** (**checkSat**() = $f$) **then**
    **return** $false$
  $L \leftarrow$**findUnitLiteral**()
  **if** ($L \neq NULL$) **then**
    **propagate**($L, level$)
    **return DPLL**($level$)
  $L \leftarrow$**pickLiteral**()
  $level \leftarrow level + 1$
  **propagate**($L, level$)
  **if** (**DPLL**($level$) $\neq false$) **then**
    **return DPLL**($level$)
  **backtrack**($level - 1$)
  **propagate**($\bar{L}, level$)
  **return DPLL**($level$)
```

```
**checkSat**()
  **if** there is a clause $C$ s.t. $C.count = 0$ **then**
    **return** $f$
  **if** for every clause $C$, $C.sat = t$ **then**
    **return** $t$
  **else**
    **return** $u$
```

```
**findUnitLiteral**()
  **for all** clauses $C$ **do**
    **if** ($C.sat = f$ **and** $C.count = 1$) **then**
      **for all** literals $L \in C$ **do**
        **if** ($L.value = u$) **then**
          **return** $L$
  **return** $NULL$
```

```
**pickLiteral**()
  **return** (a literal $L$ s.t. $L.value = u$ **and** $L \in C$ for some clause $C$ s.t. $C.sat = f$)
```

```
propagate($L, level$)
 $P \leftarrow \{L \mid L.value = t\}$
 for all literals $L'$ s.t. $P \cup \{L\} \models L'$ do
  if ($L'.value = u$) then
   $L'.value \leftarrow t$
   $L'.level \leftarrow level$
   for all clauses $C$ s.t. $L' \in C$ do
    if ($C.sat = f$) then
     $C.sat \leftarrow t$
     $C.level \leftarrow level$
 for all literals $L'$ s.t. $P \cup \{L\} \Dashv L'$ do
   if ($L'.value = u$) then
    $L'.value \leftarrow f$
    $L'.level \leftarrow level$
    for all clauses $C$ with $L' \in C$ do
     $C.count \leftarrow C.count - 1$
```

```
backtrack($level$)
 for all literals $L$ do
  if ($L.level > level$) then
   if ($L.value = t$) then
    for all clauses $C$ s.t. $L \in C$
     if ($C.level > level$) then
      $C.sat \leftarrow f$
      $C.level \leftarrow -1$
   if ($L.value = f$) then
    for all clauses $C$ s.t. $L \in C$ do
     $C.count \leftarrow C.count + 1$
   $L.value \leftarrow u$
   $L.level \leftarrow -1$
```

Solve() starts by initializing the data structures and then makes the call **DPLL**(0), which will return a partial interpretation P such that $P \models T$, if one exists; otherwise $false$ is returned.

Below are descriptions of the auxiliary functions used by **DPLL()**.

checkSat(): This function checks to see if $P \models T$, $P \Dashv T$ or otherwise. If $P \models T$ it returns t. If $P \Dashv T$ it returns f. Otherwise it returns u.

findUnitLiteral(): This function returns a unit literal w.r.t. P if one exists. Otherwise, it returns $NULL$.

pickLiteral(): This function returns an unassigned random literal from a clause not satisfied with respect to the current partial interpretation.

propagate($L, level$): This function satisfies all literals L' s.t. $P \cup \{L\} \models L'$ and $P \not\models L'$. The function also satisfies all not yet satisfied clauses C containing such L'. This function falsifies all unassigned literals L' s.t. $P \cup \{L\} \dashv L'$ and decrements $C.count$ of all clauses C s.t. $L' \in C$.

backtrack($level$): This function restores the state just before the decision that began level $level + 1$.

As usual, let's look at an example to help us understand Algorithm 3. Consider a theory T with variables $V = \{1, 2, 3\}$ such that for each $v \in V$, $dom(v) = \{0, 1, 2\}$. The theory T consists of the four clauses given below. The current partial interpretation $P = \emptyset$. **Solve()** initializes the data structures and makes the first call **DPLL(0)**.

$P = \emptyset$

$Clause1\ \{1 = 1\}$ $U(1)$

$Clause2\ \{1 = 0,\ 2 = 0,\ 3 = 1\}\, U(3)$

$Clause3\ \{1 = 0,\ 2 = 1,\ 3 = 0\}\, U(3)$

$Clause4\ \{1 = 0,\ 2 = 2,\ 3 \neq 0\}\, U(3)$

The number in brackets after the symbol U is the number of currently unfalsified literals in the clause. (That is, it is the current value of the $count$ field for that clause.)

$Clause1$ is unit, as $Clause1.sat = f$ and $Clause1.count = 1$. So $1 = 1$ is a unit literal. So literal $1 = 1$ is propagated at level 0.

$P = \{1 = 1_0\}$

$Clause1\ \{\mathit{1 = 1}\}$ $S(1)$

$Clause2\ \{~~$1 = 0$~~,\ 2 = 0,\ 3 = 1\}\, U(2)$

$Clause3$ $\{$~~$1=0$~~$,\ 2=1,\ 3=0\}\,U(2)$

$Clause4$ $\{$~~$1=0$~~$,\ 2=2,\ 3\neq 0\}\,U(2)$

The subscript 0 on literal $1=1$ in P denotes the level at which $1=1$ became satisfied with respect to the current partial interpretation P. (That is, 0 is the current value of the *level* field for the literal $1=1$.) Notice that in addition to literal $1=1$, the literals $1\neq 0$, $1\neq 2$, $1\neq 3$ also have their *value* field set to t and *level* field set to 0. (Here and throughout, when we show the current partial interpretation P, we supress the additional literals. For instance, here we show only $1=1$ in P, even though, technically speaking, the literals $1\neq 0$, $1\neq 2$ and $1\neq 3$ belong to P also.) Similarly, literals $1=0$, $1\neq 1$, $1=2$, $1=3$ have their *value* field set to f and *level* field set to 0. Now $Clause1.sat = t$ and $Clause1.level = 0$, but T is neither satisfied nor falsified and there are no more unit literals, so we pick a literal. Assume the "decision literal" is $2=0$ from $Clause2$. The *level* parameter is incremented to 1, since $2=0$ will be our first decision literal, and literal $2=0$ is propagated.

$P=\{1=1_0,\ 2=0_1\}$

$Clause1$ $\{\mathit{1=1}\}$ $S(1)$

$Clause2$ $\{$~~$1=0$~~$,\ \mathit{2=0},\ 3=1\}$ $S(2)$

$Clause3$ $\{$~~$1=0$~~, ~~$2=1$~~$,\ 3=0\}$ $U(1)$

$Clause4$ $\{$~~$1=0$~~, ~~$2=2$~~$,\ 3\neq 0\}$ $U(1)$

Now $Clause2.sat = t$ and $Clause2.level = 1$. Notice that $Clause3$ is unit, as $Clause3.sat = f$ and $Clause3.count = 1$. So we must satisfy unit literal $3=0$. So literal $3=0$ is propagated at level 1.

$P=\{1=1_0,\ 2=0_1,\ 3=0_1\}$

$Clause1$ $\{\mathit{1=1}\}$ $S(1)$

$Clause2$ $\{$~~$1=0$~~$,\ \mathit{2=0},$ ~~$3=1$~~$\}$ $S(1)$

$Clause3$ $\{$~~$1=0$~~, ~~$2=1$~~$,\ \mathit{3=0}\}$ $S(1)$

$Clause4$ $\{$~~$1=0$~~, ~~$2=2$~~, ~~$3\neq 0$~~$\}$ $F(0)$

Now $Clause3.sat = t$ and $Clause3.level = 1$. But T is falsified, as $Clause4.count = 0$, so we backtrack one level prior to current $level$ i.e. backtrack to level 0.

$P = \{1 = 1_0\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~$,\ 2 = 0,\ 3 = 1\}$ $U(2)$

$Clause3$ $\{$~~$1 = 0$~~$,\ 2 = 1,\ 3 = 0\}$ $U(2)$

$Clause4$ $\{$~~$1 = 0$~~$,\ 2 = 2,\ 3 \neq 0\}$ $U(2)$

After backtracking to level 0, the complement of the previous decision literal $2 = 0$ (i.e. $2 \neq 0$) is propagated, again at level 1.

$P = \{1 = 1_0,\ 2 \neq 0_1\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~, ~~$2 = 0$~~$,\ 3 = 1\}$ $U(1)$

$Clause3$ $\{$~~$1 = 0$~~$,\ 2 = 1,\ 3 = 0\}$ $U(2)$

$Clause4$ $\{$~~$1 = 0$~~$,\ 2 = 2,\ 3 \neq 0\}$ $U(2)$

Now $Clause2$ is unit, as $Clause2.sat = f$ and $Clause2.count = 1$. So we must satisfy unit literal $3 = 1$ in $Clause2$. Thus, literal $3 = 1$ is propagated at level 1.

$P = \{1 = 1_0,\ 2 \neq 0_1,\ 3 = 1_1\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~, ~~$2 = 0$~~$,\ \mathit{3 = 1}\}$ $S(1)$

$Clause3$ $\{$~~$1 = 0$~~$,\ 2 = 1,$ ~~$3 = 0$~~$\}$ $U(1)$

$Clause4$ $\{$~~$1 = 0$~~$,\ 2 = 2,\ \mathit{3 \neq 0}\}$ $S(2)$

Now $Clause3$ is unit, as $Clause3.sat = f$ and $Clause3.count = 1$. So we must satisfy unit literal $2 = 1$ in $Clause3$. So literal $2 = 1$ is propagated at level 1.

$P = \{1 = 1_0,\ 2 \neq 0_1,\ 3 = 1_1,\ 2 = 1_1\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~, ~~$2 = 0$~~$,\ \mathit{3 = 1}\}$ $S(1)$

$Clause3$ $\{$~~$1 = 0$~~, $2 = 1$, ~~$3 = 0$~~$\}$ $S(1)$

$Clause4$ $\{$~~$1 = 0$~~, ~~$2 = 2$~~, $3 \neq 0\}$ $S(1)$

Now the theory is satisfied, so **DPLL(1)** returns the set of literals having $L.value = t$, which is then returned by **DPLL(0)** and so by **Solve()**. This shows that $P = \{1 = 1, 2 \neq 0, 3 = 1, 2 = 1\}$ satisfies T. And since $I = \{1 = 1, 2 = 1, 3 = 1\}$ is an extension of P, we can conclude that I is a model of T.

2.4 Clause Learning and Non-Chronological Backtracking and Algorithm 4

Clause learning, along with non-chronological backtracking, was first incorporated into the SAT solvers GRASP [22] and relsat [2]. These techniques help in solving satisfiability problems that would otherwise be very slow to solve.

The basic idea behind clause learning is the addition of a clause to a theory. The learned clause is always a clause entailed by the theory, whereas clause is said to be *entailed* by a set of clauses, if any interpretation that satisfies the set of clauses satisfies the clause. So this method is sound, but addition of the learned clause can help advance the search – much as lemmas can help in the search for a proof of a theorem. The learned clause is formed taking into account the assignments due to which the theory is falsified with respect to the current partial interpretation, representing a constraint preventing the same assignments to occur in the future by generating unit literals for propagation before the theory can be falsified in that way again.

Chronological backtracking is the method we have used so far for continuing the search after encountering a conflict. In non-chronological backtracking, we choose the level to backtrack to in some other way. Most commonly, we undo more assignments than we would in chronological backtracking.

In this section we will describe how clause learning and non-chronological backtracking work in principle, with the help of an iterative version of the **DPLL** function used in Algorithm 3. We will not specify in detail our algorithms for clause learning and non-chronological backtracking, because those details are not especially relevant to our discussion at this point. But it is important to note that the backtracking level should always be prior to the current decision level, since we need to undo some part of the current falsifying assignment. In the current implementation it is the second highest level among the literals in the learned clause, or one less than the current level if the learned clause only consists of one literal. This is one way of deciding the backtracking level but there are other possible choices.

There are global variables T and $level$ for storing the theory and the current decision level.

Algorithm 4

```
Solve()
 for all literals $L$ do
  $L.value \leftarrow u$
  $L.level \leftarrow -1$
 for all clauses $C$ do
  $C.sat \leftarrow f$
  $C.count \leftarrow$ number of literals in $C$
  $C.level \leftarrow -1$
 DPLL()
```

```
T is a theory
DPLL()
 $level \leftarrow 0$
 while(true)
  if (checkSat() = $t$) then
   return $\{L \mid L.value = t\}$
  if (checkSat() = $f$) then
    if ($level = 0$) then
     return $false$
    else
     $learnedClause \leftarrow$ analyzeConflict()
     $level \leftarrow$ findBacktrackingLevel($learnedClause$)
     $T \leftarrow T \cup \{learnedClause\}$
     backtrack($level$)
  $L \leftarrow$findUnitLiteral()
  if ($L \neq NULL$) then
   propagate($L$, $level$)
  else
   $L \leftarrow$pickLiteral()
   $level \leftarrow level + 1$
   propagate($L$, $level$)
```

Everything works as in Algorithm 3 until the theory becomes falsified with respect to the current partial interpretation. At that point, if the theory is falsified with respect to the current partial interpetation at a level not equal to zero, then a clause is learned and added to the theory and backtracking is done to a level based on the learned clause (a form of non-chronological backtracking). The details depend on specific properties of **analyzeConflict**() and **findBacktrackingLevel**($learnedClause$), which are not given here. (Our versions of these functions are described in more detail in Chapter 5.) If the theory becomes falsified with respect to the current partial interpretation when level = 0, we return *false*, as the theory is unsatisfiable.

Let's look at few examples to see how Algorithm 4 works.

Example 1

Consider the same theory T as in Algorithm 3 with variables $V = \{1,2,3\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$. The theory T consists of the four clauses given below. The current partial interpretation $P = \emptyset$. **Solve()** initializes the data structures and makes the first call **DPLL()**.

$P = \emptyset$

$Clause1$ $\{1 = 1\}$ $U(1)$

$Clause2$ $\{1 = 0, 2 = 0, 3 = 1\}$ $U(3)$

$Clause3$ $\{1 = 0, 2 = 1, 3 = 0\}$ $U(3)$

$Clause4$ $\{1 = 0, 2 = 2, 3 \neq 0\}$ $U(3)$

Everything works same as in Algorithm 3 version of the example till the theory is falsified w.r.t. P.

$P = \{1 = 1_0,\ 2 = 0_1,\ 3 = 0_1\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~$,\ \mathit{2 = 0},$ ~~$3 = 1$~~$\}$ $S(1)$

$Clause3$ $\{$~~$1 = 0$~~$,$ ~~$2 = 1$~~$,\ \mathit{3 = 0}\}$ $S(1)$

$Clause4$ $\{$~~$1 = 0$~~$,$ ~~$2 = 2$~~$,$ ~~$3 \neq 0$~~$\}$ $F(0)$

Currently T is falsified, as $Clause4.count = 0$. As current level is not equal to 0, $Clause5$ is learned and added to the theory and we backtrack to level 0 based on the learned $Clause5$.

$P = \{1 = 1_0\}$

$Clause1$ $\{\mathit{1 = 1}\}$ $S(1)$

$Clause2$ $\{$~~$1 = 0$~~$,\ 2 = 0,\ 3 = 1\}$ $U(2)$

$Clause3$ $\{$~~$1 = 0$~~$,\ 2 = 1,\ 3 = 0\}$ $U(2)$

$Clause4$ $\{$~~$1 = 0$~~$,\ 2 = 2,\ 3 \neq 0\}$ $U(2)$

$Clause5$ $\{$~~$1 = 0$~~$,\ 2 \neq 0\}$ $U(1)$

After backtracking to level 0, $Clause5$ is unit, as $Clause5.sat = f$ and $Clause5.count = 1$. So we must satisfy unit literal $2 \neq 0$ in $Clause5$. Thus, literal $2 \neq 0$ is propagated at level

0.

$P = \{1 = 1_0,\ 2 \neq 0_0\}$

$Clause1\ \{\mathit{1 = 1}\}$ $S(1)$

$Clause2\ \{\cancel{1 = 0},\ \cancel{2 = 0},\ 3 = 1\}\ U(1)$

$Clause3\ \{\cancel{1 = 0},\ 2 = 1,\ 3 = 0\}\ U(2)$

$Clause4\ \{\cancel{1 = 0},\ 2 = 2,\ 3 \neq 0\}\ U(2)$

$Clause5\ \{\cancel{1 = 0},\ \mathit{2 \neq 0}\}$ $S(1)$

Now $Clause2$ is unit, as $Clause2.sat = f$ and $Clause2.count = 1$. So we must satisfy unit literal $3 = 1$ in $Clause2$. Thus, literal $3 = 1$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 \neq 0_0,\ 3 = 1_0\}$

$Clause1\ \{\mathit{1 = 1}\}$ $S(1)$

$Clause2\ \{\cancel{1 = 0},\ \cancel{2 = 0},\ \mathit{3 = 1}\}\ S(1)$

$Clause3\ \{\cancel{1 = 0},\ 2 = 1,\ \cancel{3 = 0}\}\ U(1)$

$Clause4\ \{\cancel{1 = 0},\ 2 = 2,\ \mathit{3 \neq 0}\}\ S(2)$

$Clause5\ \{\cancel{1 = 0},\ \mathit{2 \neq 0}\}$ $S(1)$

Now $Clause3$ is unit, as $Clause3.sat = f$ and $Clause3.count = 1$. So we must satisfy unit literal $2 = 1$ in $Clause3$. So literal $2 = 1$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 \neq 0_0,\ 3 = 1_0, 2 = 1_0\}$

$Clause1\ \{\mathit{1 = 1}\}$ $S(1)$

$Clause2\ \{\cancel{1 = 0},\ \cancel{2 = 0},\ \mathit{3 = 1}\}\ S(1)$

$Clause3\ \{\cancel{1 = 0},\ \mathit{2 = 1},\ \cancel{3 = 0}\}\ S(1)$

$Clause4\ \{\cancel{1 = 0},\ \cancel{2 = 2},\ \mathit{3 \neq 0}\}\ S(1)$

$Clause5\ \{\cancel{1 = 0},\ \mathit{2 \neq 0}\}$ $S(1)$

Now the theory is satisfied, so **DPLL()** returns the current partial interpretation $\{1 = 1, 2 \neq 0, 3 = 1, 2 = 1\}$, and so does **Solve()**. This shows that $P = \{1 = 1, 2 \neq 0, 3 = 1, 2 = 1\}$ satisfies T. And since $I = \{1 = 1, 2 = 1, 3 = 1\}$ is an extension of P, we can conclude that

I is a model of T.

Example 2

Consider a theory T with variables $V = \{1,2\}$ such that for each $v \in V$, $dom(v) = \{0,1\}$. The theory T consists of the four clauses given below. The current partial interpretation $P = \emptyset$. **Solve()** initializes the data structures and makes the first call **DPLL()**.

$P = \emptyset$

$Clause1$ $\{1 = 0,\ 2 = 0\}$ $U(2)$

$Clause2$ $\{1 \neq 0,\ 2 = 0\}$ $U(2)$

$Clause3$ $\{1 = 0,\ 2 \neq 0\}$ $U(2)$

$Clause4$ $\{1 \neq 0,\ 2 \neq 0\}$ $U(2)$

T is neither satisfied nor falsified and there are no unit literals, so we pick a literal. Assume the "decision literal" is $1 = 0$ from $Clause1$. The $level$ parameter is incremented to 1, since $1 = 0$ will be our first decision literal, and literal $1 = 0$ is propagated.

$P = \{1 = 0_1\}$

$Clause1$ $\{\mathit{1} = \mathit{0},\ 2 = 0\}$ $S(2)$

$Clause2$ $\{$~~$1 \neq 0$~~$,\ 2 = 0\}$ $U(1)$

$Clause3$ $\{\mathit{1} = \mathit{0},\ 2 \neq 0\}$ $S(2)$

$Clause4$ $\{$~~$1 \neq 0$~~$,\ 2 \neq 0\}$ $U(1)$

Now $Clause1.sat = t$, $Clause1.level = 1$ and $Clause3.sat = t$, $Clause3.level = 1$. Notice that $Clause2$ is unit, as $Clause2.sat = f$ and $Clause2.count = 1$. So we must satisfy unit literal $2 = 0$. So literal $2 = 0$ is propagated at level 1.

$P = \{1 = 0_1,\ 2 = 0_1\}$

$Clause1$ $\{\mathit{1} = \mathit{0},\ \mathit{2} = \mathit{0}\}$ $S(2)$

$Clause2$ $\{$~~$1 \neq 0$~~$,\ \mathit{2} = \mathit{0}\}$ $S(1)$

$Clause3$ $\{\mathit{1} = \mathit{0},$ ~~$2 \neq 0$~~$\}$ $S(1)$

$Clause4$ $\{$~~$1 \neq 0$~~$,$ ~~$2 \neq 0$~~$\}$ $F(0)$

Now $Clause2.sat = t$ and $Clause2.level = 1$. But T is falsified, as $Clause4.count = 0$. As current level is not equal to 0, $Clause5$ is learned and added to the theory and we backtrack to level 0 based on learned $Clause5$.

$P = \emptyset$

$Clause1\ \{1 = 0,\ 2 = 0\}\ \ U(2)$

$Clause2\ \{1 \neq 0,\ 2 = 0\}\ \ U(2)$

$Clause3\ \{1 = 0,\ 2 \neq 0\}\ \ U(2)$

$Clause4\ \{1 \neq 0,\ 2 \neq 0\}\ \ U(2)$

$Clause5\ \{1 \neq 0\}\qquad\quad U(1)$

After backtracking to level 0, $Clause5$ is unit, as $Clause5.sat = f$ and $Clause5.count = 1$. So we must satisfy unit literal $1 \neq 0$ in $Clause5$. Thus, literal $1 \neq 0$ is propagated at level 0.

$P = \{1 \neq 0_0\}$

$Clause1$ {~~$1 = 0$~~, $2 = 0$} $U(1)$

$Clause2\ \{\mathit{1 \neq 0},\ 2 = 0\}\ \ S(2)$

$Clause3$ {~~$1 = 0$~~, $2 \neq 0$} $U(1)$

$Clause4\ \{\mathit{1 \neq 0},\ 2 \neq 0\}\ \ S(2)$

$Clause5\ \{\mathit{1 \neq 0}\}\qquad\quad S(1)$

Now $Clause2.sat = t$, $Clause2.level = 0$ and $Clause4.sat = t$, $Clause4.level = 0$. Notice that $Clause1$ is unit, as $Clause1.sat = f$ and $Clause1.count = 1$. So we must satisfy unit literal $2 = 0$. So literal $2 = 0$ is propagated at level 0.

$P = \{1 \neq 0_0,\ 2 = 0_0\}$

$Clause1$ {~~$1 = 0$~~, $\mathit{2 = 0}$} $S(1)$

$Clause2\ \{\mathit{1 \neq 0},\ \mathit{2 = 0}\}\ \ S(2)$

$Clause3$ {~~$1 = 0$~~, ~~$2 \neq 0$~~} $F(0)$

$Clause4$ {$\mathit{1 \neq 0}$, ~~$2 \neq 0$~~} $S(1)$

$Clause5$ $\{\mathit{1} \neq \mathit{0}\}$ $S(1)$

Now $Clause1.sat = t$ and $Clause1.level = 0$. But T is falsified, as $Clause3.count = 0$. As the theory is falsified with respect to the current partial interpretation when level equals 0, we return *false*, implying the theory is unsatisfiable.

2.5 Problem with Lal's FD-SAT Solver

Algorithm 4 is essentially the algorithm implemented by Lal [15]. Lal obtained very promising results and we were unable to replicate them. In trying to understand the reason we discovered an implementation bug in the clause learning mechanism. The solver produced learned clauses that are different (and stronger!) than those given by his algorithm. In some cases, this resulted in learning of clauses which are not entailed by the theory, thereby making the solver unsound. Consider a theory T with variables $V = \{1,2,3,4\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$. The theory T consists of the eighteen clauses given below.

$Clause1$ $\{1 \neq 0,\ 2 \neq 0\}$

$Clause2$ $\{1 \neq 0,\ 3 \neq 0\}$

$Clause3$ $\{1 \neq 0,\ 4 \neq 0\}$

$Clause4$ $\{2 \neq 0,\ 3 \neq 0\}$

$Clause5$ $\{2 \neq 0,\ 4 \neq 0\}$

$Clause6$ $\{3 \neq 0,\ 4 \neq 0\}$

$Clause7$ $\{1 \neq 1,\ 3 \neq 1\}$

$Clause8$ $\{1 \neq 1,\ 4 \neq 1\}$

$Clause9$ $\{2 \neq 1,\ 3 \neq 1\}$

$Clause10$ $\{2 \neq 1,\ 4 \neq 1\}$

$Clause11$ $\{3 \neq 1,\ 4 \neq 1\}$

$Clause12$ $\{1 \neq 2,\ 2 \neq 2\}$

$Clause13$ $\{1 \neq 2,\ 3 \neq 2\}$

$Clause14$ $\{1 \neq 2,\ 4 \neq 2\}$

$Clause15$ $\{2 \neq 2,\ 3 \neq 2\}$

$Clause16$ $\{2 \neq 2,\ 4 \neq 2\}$

$Clause17$ $\{3 \neq 2,\ 4 \neq 2\}$

$Clause18$ $\{1 \neq 1,\ 1 = 0,\ 1 = 1,\ 1 = 2\}$

Looking more closely, we notice that the theory T is similar to the unsatisfiable Pigeonhole Problem [18] requiring 4 pigeons to be fit in 3 holes, with a missing clause $\{1 \neq 1,\ 2 \neq 1\}$ and an additional clause $\{1 \neq 1,\ 1 = 0,\ 1 = 1,\ 1 = 2\}$. The removal of the clause $\{1 \neq 1,\ 2 \neq 1\}$ from the Pigeonhole Problem makes the theory T satisfiable, by allowing pigeons 1 and 2 to fit in the same hole, 1. Pigeons 3 and 4 can fit in holes 0 and 2 interchangeably. Thus T is satisfiable by the interpretations $\{1 = 1,\ 2 = 1,\ 3 = 0,\ 4 = 2\}$ and $\{1 = 1,\ 2 = 1,\ 3 = 2,\ 4 = 0\}$. $Clause18$ is a additional clause entailed by the theory (any interpretation that satisfies the theory also satisfies $Clause18$) so that heuristic used by the solver picks $1 = 0$ as the first decision literal.

Lal's solver has $P = \{1 = 0,\ 2 \neq 0,\ 3 \neq 0,\ 4 \neq 0,\ 2 = 1,\ 3 \neq 1,\ 3 = 2,\ 4 \neq 1,\ 4 = 2\}$ when T is first falsified, as $Clause17.count = 0$. As current level is not equal to 0, a clause is learned at this point. Lal's solver learns the clause $\{2 \neq 1\}$!

Notice that this clause is not entailed by the theory, and, in fact, its addition makes the theory unsatisfiable.

In general, the learned clause should be entailed by the theory. But typically we can say more: the learned clause should be entailed by the falsified clause plus the clauses that became unit at the current level. Indeed, we uncovered the implementation bug in Lal's solver by considering this stronger property. We found that we could not duplicate Lal's remarkably good solution times on problems that required lots of backtracking, and we noticed that his solver required many fewer backtracks than other solvers, when run on hard unsatisfiable problems. Of course, when T is unsatisfiable, every clause is entailed by T, so no learned clause can fail to be entailed by T. But we soon found examples of learned clauses that were not entailed by the falsified clause and the current unit clauses. This allowed us to construct the example above, showing that Lal's solver is unsound.

And since this implementation bug also has the effect of short-circuiting the search on hard unsatisfiable problems, it invalidates Lal's experimental results on such problems.

Chapter 3

Watched Literals and Algorithm 5

The adoption of clause learning and non-chronological backtracking can greatly improve the performance of satisfiability solvers. One possible way to improve the performance further would be to reduce the time spent by the solvers while backtracking. One way to reduce time while backtracking is to use an alternative approach for detecting unit literals, satisfied clauses and falsified clauses (with respect to the partial interpretation) known as "watched literals" [18]. The watched literals technique improves efficiency while backtracking and may result in improving overall efficiency.

In watched literals we maintain two "watched literal" pointers with each clause. At all times, the two watched literal pointers point to distinct literals in the clause, if there are at least two literals. If there is only one literal in the clause, then the first pointer points to it while the second pointer is *NULL*. (We assume there are no empty clauses in the theory.) From then on, we maintain the following four properties. (1) A clause is satisfied iff its first watched literal is satisfied. (2) A clause is falsified iff its first watched literal is falsified and the second watched literal is either falsified or *NULL*. (3) A clause is unit iff it has one of its watched literals falsified or *NULL* and the other unassigned. (4) A clause is unassigned and not unit iff both its watched literals are unassigned.

So when a clause becomes satisfied, the first watched literal pointer is set to point to the satisfied literal in the clause. (And if it happens that the second watched literal is the one being satisfied, we swap the watched literal pointers.) If the clause is not yet satisfied and one of the watched literals is becoming false, we need to do some watched literal maintenance work. Suppose the watched literal other than the currently falsified watched literal already points to a falsified literal or is *NULL*. Then we have a falsified clause with respect to the current partial interpretation, and nothing need be done. Otherwise, if the watched literal other than the currently falsified watched literal is unassigned then we need to update the current falsified watched literal to point to an unassigned literal in the clause different than the other watched literal, if one is available. If none is available, then we have a unit clause with unit literal the other watched literal, and nothing need be done.

In Algorithm 5, we will present the watched literals version of Algorithm 4. We still maintain global data structures for representing the current partial interpretation and keeping track of our progress toward satisfying the theory, but now using watched literals for some of the work.

Each literal L has again two associated fields, *L.value* and *L.level*, and they work the same as in Algorithm 4.

A clause C now has two associated fields: $C.W1$, and $C.W2$. If the clause C consists of only one literal then $C.W1$ points to the only literal in the clause and $C.W2$ is *NULL* (null pointer). Otherwise, $C.W1$ and $C.W2$ always point to distinct literals in the clause C. The watched literal $C.W1$ will be used to keep track of whether C is currently satisfied; C is satisfied iff $C.W1.value = t$. Moreover, if C is satisfied, we can now obtain the level at which it became satisfied by looking at $C.W1.level$. This helps a lot while backtracking, as we no longer have to un-set any *sat* flag and *level* field associated with the clause, since the $W1's\ value$ and *level* fields take care of this. This improves efficiency while backtracking and may improve efficiency overall. We also no longer maintain a *count* field for each clause.

Algorithm 5

```
Solve()
 for all literals $L$ do
  $L.value \leftarrow u$
  $L.level \leftarrow -1$
 for all clauses $C$ do
  $C.W1 \leftarrow$ points to the first literal in clause $C$
  $C.W2 \leftarrow$ points to the second literal in clause $C$, otherwise $NULL$
 DPLL()
```

```
checkSat()
 if there is a clause $C$ s.t. $C.W1.value = f$ and
 $(C.W2.value = f$ or $C.W2.value = NULL)$ then
  return $f$
 else if for every clause $C$, $C.W1.value = t$ then
  return $t$
 else
  return $u$
```

```
findUnitLiteral()
 for all clauses $C$ do
  if $(C.W1.value = u)$ then
   if $(C.W2.value = NULL$ or $C.W2.value = f)$ then
    return $C.W1$
  if $(C.W2.value = u)$ then
   if $(C.W1.value = f)$ then
    return $C.W2$
 return $NULL$
```

```
pickLiteral()
 return (a literal $L$ s.t. $L.value = u$ and for some clause $C$, $L \in C$ s.t. $C.W1.value = u$)
```

```
**propagate**($L, level$)
 $P \leftarrow \{L \mid L.value = t\}$
 for all literals $L'$ s.t. $P \cup \{L\} \models L'$ do
  if ($L'.value = u$) then
   $L'.value \leftarrow t$
   $L'.level \leftarrow level$
   for all clauses $C$ s.t. $L' \in C$ do
    if ($C.W1.value \neq t$) then
     if ($C.W2 \neq L'$) then
      $C.W1 \leftarrow L'$
     else
      $C.W2 \leftarrow C.W1$
      $C.W1 \leftarrow L'$
 for all literals $L'$ s.t. $P \cup \{L\} \dashv L'$ do
  if ($L'.value = u$) then
   $L'.value \leftarrow f$
   $L'.level \leftarrow level$
   for all clauses $C$ with $L' \in C$ do
    if ($C.W1.value \neq t$) then
     if ($C.W1 = L'$ or $C.W2 = L'$) then
      swapPointer($C$)
```

```
swapPointer($C$)
 if ($C.W2.value = u$) then
  if there exists $L \in C$ s.t. $L \neq C.W2$ and $L.value = u$ then
   $C.W1 \leftarrow L$
 if ($C.W1.value = u$) then
  if there exists $L \in C$ s.t. $L \neq C.W1$ and $L.value = u$ then
   $C.W2 \leftarrow L$
```

```
backtrack($level$)
 for all literals $L$ do
  if ($L.level > level$) then
   $L.value \leftarrow u$
   $L.level \leftarrow -1$
```

As before, **Solve()** starts with initializing the data structures and makes a call **DPLL()**. Please note that **DPLL()** is the same as in algorithm 4. Again, the call **DPLL()** will return the partial interpretation based on the properties of **analyzeConflict()** and **findBacktrackingLevel**(*learnedClause*); otherwise *false* is returned.

Most auxiliary functions perform the same function as in Algorithm 3 but a bit differently as we see below:

checkSat(): This function checks to see if $P \models T$, $P \dashv T$ or otherwise. If $P \models T$ it returns t. If $P \dashv T$ it returns f. Otherwise it returns u. This is now accomplished by looking at the watched literals.

findUnitLiteral(): This function returns a unit literal w.r.t. P if one exists. Otherwise, it returns $NULL$. Again, this is done by looking at watched literals.

pickLiteral(): This function returns an unassigned random literal from a clause not satisfied with respect to the current partial interpretation. Here too watched literals are used.

propagate($L, level$**):** This function satisfies all literals L' s.t. $P \cup \{L\} \models L'$ and $P \not\models L'$. The function also satisfies all not yet satisfied clauses C containing such L' with respect to P, and then makes sure the newly satisfied literal is $C.W1$. This function falsifies unassigned literals L' s.t. $P \cup \{L\} \dashv L'$ and for all not yet satisfied clauses C having L' as one of their watched literals calls function **swapPointer(**C**).**

swapPointer(C**):** If one of the watched literals in clause C is unassigned and other is falsified then the function tries to find an unassigned literal replacement for the falsified literal other than the already unassigned watched literal.

backtrack($level$**):** This function restores the state just before the decision that began level $level+1$. Notice that, on comparison, the **backtrack()** function in Algorithm 5 does almost no work compared to the one in Algorithm 4.

Let's look at the Example 1 used in Algorithm 4 now solved using Algorithm 5!

As before, the current partial interpretation $P = \emptyset$. **Solve()** initializes the data structures and make the first call **DPLL()**.

$P = \emptyset$

$Clause1$	$\{1 = 1\}$	U	$W1(1 = 1)$	$W2(NULL)$
$Clause2$	$\{1 = 0,\ 2 = 0,\ 3 = 1\}$	U	$W1(1 = 0)$	$W2(2 = 0)$
$Clause3$	$\{1 = 0,\ 2 = 1,\ 3 = 0\}$	U	$W1(1 = 0)$	$W2(2 = 1)$
$Clause4$	$\{1 = 0,\ 2 = 2,\ 3 \neq 0\}$	U	$W1(1 = 0)$	$W2(2 = 2)$

$W1$ and $W2$ after the symbol U represent the watched literals in a clause.

$Clause1$ is unit as $Clause1.W1.value = u$ and $Clause1.W2.value = NULL$, so $1 = 1$ is a unit literal. So literal $1 = 1$ is propagated at level 0.

$P = \{1 = 1_0\}$

$Clause1$	$\{\mathit{1 = 1}\}$	S	$W1(\mathit{1 = 1})$	$W2(NULL)$
$Clause2$	$\{\cancel{1 = 0},\ 2 = 0,\ 3 = 1\}$	U	$W1(3 = 1)$	$W2(2 = 0)$
$Clause3$	$\{\cancel{1 = 0},\ 2 = 1,\ 3 = 0\}$	U	$W1(3 = 0)$	$W2(2 = 1)$
$Clause4$	$\{\cancel{1 = 0},\ 2 = 2,\ 3 \neq 0\}$	U	$W1(3 \neq 0)$	$W2(2 = 2)$

Literal $1 = 0$ is falsified. $Clause2$, $Clause3$ and $Clause4$ are not satisfied and have literal $1 = 0$ as their first watched literal, so watched literal maintenance is done and the first watched literal in these three clauses is replaced by an unassigned literal. Thus $Clause2.W1$ becomes $3 = 1$, $Clause3.W1$ becomes $3 = 0$ and $Clause4.W1$ becomes $3 \neq 0$.

Now T is neither satisfied nor falsified and there are no more unit literals, so we pick a literal. Assume the picked literal is $2 = 0$ from $Clause2$. The $level$ is incremented to 1 and literal $2 = 0$ is propagated.

$P = \{1 = 1_0,\ 2 = 0_1\}$

$Clause1$	$\{\mathit{1 = 1}\}$	S	$W1(\mathit{1 = 1})$	$W2(NULL)$
$Clause2$	$\{\cancel{1 = 0},\ \mathit{2 = 0},\ 3 = 1\}$	S	$W1(\mathit{2 = 0})$	$W2(3 = 1)$
$Clause3$	$\{\cancel{1 = 0},\ \cancel{2 = 1},\ 3 = 0\}$	U	$W1(3 = 0)$	$W2(\cancel{2 = 1})$
$Clause4$	$\{\cancel{1 = 0},\ \cancel{2 = 2},\ 3 \neq 0\}$	U	$W1(3 \neq 0)$	$W2(\cancel{2 = 2})$

Now $Clause3$ is unit as $Clause3.W1.value = u$ and $Clause3.W2.value = f$. So we must satisfy unit literal $3 = 0$ in *Clause*3. So literal $3 = 0$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 = 0_1,\ 3 = 0_1\}$

*Clause*1 $\{1 = 1\}$	S	$W1(1 = 1)$	$W2(NULL)$
*Clause*2 $\{$~~$1 = 0$~~$,\ 2 = 0,\ $~~$3 = 1$~~$\}$	S	$W1(2 = 0)$	$W2($~~$3 = 1$~~$)$
*Clause*3 $\{$~~$1 = 0$~~$,\ $~~$2 = 1$~~$,\ 3 = 0\}$	S	$W1(3 = 0)$	$W2($~~$2 = 1$~~$)$
*Clause*4 $\{$~~$1 = 0$~~$,\ $~~$2 = 2$~~$,\ $~~$3 \neq 0$~~$\}$	F	$W1($~~$3 \neq 0$~~$)$	$W2($~~$2 = 2$~~$)$

Here T is falsified because $Clause4.W1.value = f$ and $Clause4.W2.value = f$. As current level is not equal to 0, *Clause*5 is learned and added to the theory and we backtrack to level 0 based on learned *Clause*5.

$P = \{1 = 1_0\}$

*Clause*1 $\{1 = 1\}$	S	$W1(1 = 1)$	$W2(NULL)$
*Clause*2 $\{$~~$1 = 0$~~$,\ 2 = 0,\ 3 = 1\}$	U	$W1(2 = 0)$	$W2(3 = 1)$
*Clause*3 $\{$~~$1 = 0$~~$,\ 2 = 1,\ 3 = 0\}$	U	$W1(3 = 0)$	$W2(2 = 1)$
*Clause*4 $\{$~~$1 = 0$~~$,\ 2 = 2,\ 3 \neq 0\}$	U	$W1(3 \neq 0)$	$W2(2 = 2)$
*Clause*5 $\{$~~$1 = 0$~~$,\ 2 \neq 0\}$	U	$W1(2 \neq 0)$	$W2($~~$1 = 0$~~$)$

After backtracking to level 0, *Clause*5 is unit because $Clause5.W1.value = u$ and *Clause5.W2.value = f.* So we must satisfy unit literal $2 \neq 0$ in *Clause*5. Thus, literal $2 \neq 0$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 \neq 0_0\}$

*Clause*1 $\{1 = 1\}$	S	$W1(1 = 1)$	$W2(NULL)$
*Clause*2 $\{$~~$1 = 0$~~$,\ $~~$2 = 0$~~$,\ 3 = 1\}$	U	$W1($~~$2 = 0$~~$)$	$W2(3 = 1)$
*Clause*3 $\{$~~$1 = 0$~~$,\ 2 = 1,\ 3 = 0\}$	U	$W1(3 = 0)$	$W2(2 = 1)$
*Clause*4 $\{$~~$1 = 0$~~$,\ 2 = 2,\ 3 \neq 0\}$	U	$W1(3 \neq 0)$	$W2(2 = 2)$
*Clause*5 $\{$~~$1 = 0$~~$,\ 2 \neq 0\}$	S	$W1(2 \neq 0)$	$W2($~~$1 = 0$~~$)$

Now *Clause*2 is unit as $Clause2.W1.value = f$ and $Clause2.W2.value = u$. So we must satisfy unit literal $3 = 1$ in *Clause*2. So literal $3 = 1$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 \neq 0_0,\ 3 = 1_0\}$

$Clause1\ \{1 = 1\}$	S	$W1(1 = 1)$	$W2(NULL)$
$Clause2\ \{$~~$1 = 0$~~, ~~$2 = 0$~~, $3 = 1\}$	S	$W1(3 = 1)$	$W2($~~$2 = 0$~~$)$
$Clause3\ \{$~~$1 = 0$~~, $2 = 1$, ~~$3 = 0$~~$\}$	U	$W1($~~$3 = 0$~~$)$	$W2(2 = 1)$
$Clause4\ \{$~~$1 = 0$~~, $2 = 2,\ 3 \neq 0\}$	S	$W1(3 \neq 0)$	$W2(2 = 2)$
$Clause5\ \{$~~$1 = 0$~~, $2 \neq 0\}$	S	$W1(2 \neq 0)$	$W2($~~$1 = 0$~~$)$

Literal $3 = 1$ is satisfied. $Clause2$ has the first watched literal $2 = 0$ falsified and second watched literal $3 = 1$ satisfied, so watched literal maintenance is done and the watched literal pointers are exchanged.

As we see, $Clause3$ is unit, as $Clause3.W1.value = f$ and $Clause3.W2.value = u$. So we must satisfy unit literal $2 = 1$ in $Clause3$. So, literal $2 = 1$ is propagated at level 0.

$P = \{1 = 1_0,\ 2 \neq 0_0,\ 3 = 1_0,\ 2 = 1_0\}$

$Clause1\ \{1 = 1\}$	S	$W1(1 = 1)$	$W2(NULL)$
$Clause2\ \{$~~$1 = 0$~~, ~~$2 = 0$~~, $3 = 1\}$	S	$W1(3 = 1)$	$W2($~~$2 = 0$~~$)$
$Clause3\ \{$~~$1 = 0$~~, $2 = 1$, ~~$3 = 0$~~$\}$	S	$W1(2 = 1)$	$W2($~~$3 = 0$~~$)$
$Clause4\ \{$~~$1 = 0$~~, ~~$2 = 2$~~, $3 \neq 0\}$	S	$W1(3 \neq 0)$	$W2($~~$2 = 2$~~$)$
$Clause5\ \{$~~$1 = 0$~~, $2 \neq 0\}$	S	$W1(2 \neq 0)$	$W2($~~$1 = 0$~~$)$

Literal $2 = 1$ is satisfied. $Clause3$ has the first watched literal $3 = 0$ falsified and second watched literal $2 = 1$ satisfied, so watched literal maintenance is done and the watched literal pointers are exchanged.

So **DPLL()** returns $\{1 = 1, 2 \neq 0, 3 = 1, 2 = 1\}$. This shows that $P = \{1 = 1, 2 \neq 0, 3 = 1, 2 = 1\}$ satisfies T. And since $I = \{1 = 1, 2 = 1, 3 = 1\}$ is an extension of P, we can conclude that I is a model of T.

3.1 Comparisons and Conclusions

The adoption of the watched literals approach in Algorithm 5 should improve the performance while backtracking compared to Algorithm 4 and also may improve the overall performance of the solver. We ran a set of comparisons designed to explore this question. We compared the speed of implementations of Algorithm 4 and Algorithm 5. From our comparisons we found that the watched literals approach improves the performance while backtracking on almost all sorts of problems, as expected. We also found that the watched literals approach improves the *overall* performance only on hard problems that require lots of backtracking; otherwise the bookkeeping approach wins over the watched literals approach. The bookeeping approach wins over the watched literals approach on smaller sized problems, as the **propagate**() in Algorithm 5 requires more work compared Algorithm 4 and also, there is slightly more work required by Algorithm 5 to check for unit literals. The watched literals approach wins over bookkeeping approach on harder problems requiring lots of backtracking, as the **backtracking()** function in Algorithm 5 does almost no work compared to Algorithm 4.

A note about methodology. To compare the speed of implementations of the two solvers, we implemented the algorithms as deterministic because we wanted them to execute the same search on each problem. We also attempted to keep the two implementations as similar as possible, except for the the differences required to switch from bookkeeping to watched literals.

Chapter 4

Heuristics

Up to this point in our account, if $P \not\models T$ and $P \not\models\!\!| T$ and there are no unit literals w.r.t. P, then **pickLiteral(P)** picks a literal to advance the search for a partial interpretation satisfying T. Indeed, **pickLiteral(P)** always returns a literal that will satisfy at least one additional clause when added to P. In general though this is not necessary. It is sufficient to require only that **pickLiteral(P)** return a literal L such that $P \not\models L$ and $P \not\models\!\!| L$. That is, L should be currently unassigned. (It follows that both $P \cup \{L\}$ and $P \cup \{\bar{L}\}$ are partial interpretations.) But in practice, it is probably useful to pick literals more carefully than this, using a "heuristic" to guide the search for a satisfying partial interpretation as the choice of decision literal can greatly affect performance of the algorithm. As an example, it might be useful to base the heuristic on statistics such as the *number* of clauses newly satisfied as a result of adding literal L to the current partial interpretation P, or the number of literal occurrences (in clauses not satisfied by P) that will be falsified by adding L to P (which can help produce unit clauses). Many such heuristics have been investigated in Boolean SAT solvers.

In the comparison of our bookkeeping and watched literal solvers (described at the end of the previous chapter) we used a heuristic called VSIDS, introduced in the landmark

Boolean SAT solver Chaff [18]. VSIDS stands for *Variable State Independent Decaying Sum* and works as follows: (1) Each literal has an associated counter, initially set to the number of its occurrences in the theory. (2) When a clause is learned and added to the theory, the count associated with all the literals in the clause is incremented by 1. (3) A unassigned literal from a not yet satisfied clause having a maximal count is picked as the decision literal. (4) Ties are broken randomly. (5) The literal counts for all the literals in the theory are divided by 2 after each learned clause is added to theory. VSIDS is computationally cheap and widely accepted in the literature to be a good heuristic. (To be more precise, in the comparison of bookkeeping and watched literals described in the previous chapter, we used the VSIDS heuristic, but eliminated the nondeterminism, in order to ensure that both solvers executed exactly the same search through the space of partial interpretations.)

4.1 Bookkeeping Heuristics and Algorithm 6

It might be useful to pick literals using a heuristic based on *current* counts of occurrences in unsatisfied clauses, or other similar statistics. In this section we consider an extension of the bookkeeping approach (Algorithm 4) that keeps track of this statistic, and so supports heuristics that involve the *number* of clauses newly satisfied as a result of adding literal L to the current partial interpretation P. Using this statistic we can also compute the number of literal occurrences that will be falsified in not yet satisfied clauses. Also, we can easily determine a lower bound on the number of unit clauses that will be produced. (We can look at the number of not yet satisfied clauses with count 2 in which there is an occurrence of a literal we would falsify.)

Each literal L will have several associated fields: *value*, *count* and $level$. Fields *L.value* and *L.level* work the same as in Algorithm 4. Field *L.count* is the number of clauses C

such that $L \in C$ and $P \not\models C$.

So initially for each literal L, $L.value = u$, $L.count$ is the number of clauses in which L occurs, and $L.level$ is set to -1 (a dummy value).

Now, let's look at the algorithm. Function **DPLL()** is exactly as in Algorithm 4, and so is not shown here.

Algorithm 6

```
Solve()
 for all literals $L$ do
  $L.value \leftarrow u$
  $L.count \leftarrow$ number of clauses containing $L$
  $L.level \leftarrow -1$
 for all clauses $C$ do
  $C.sat \leftarrow f$
  $C.count \leftarrow$ number of literals in $C$
  $C.level \leftarrow -1$
 DPLL()
```

```
pickLiteral()
 $max = 0$
 for all literals $L$ do
  if ($L.value = u$ and $L.count > max$) then
   $L' \leftarrow L$
   $max \leftarrow L.count$
 return $L'$
```

```
propagate($L, level$)
 $P \leftarrow \{L \mid L.value = t\}$
 for all literals $L'$ s.t. $P \cup \{L\} \models L'$ do
  if ($L'.value = u$) then
   $L'.value \leftarrow t$
   $L'.level \leftarrow level$
   for all clauses $C$ s.t. $L' \in C$ do
    if ($C.sat = f$) then
     $C.sat \leftarrow t$
     $C.level \leftarrow level$
     for all literals $L'' \in C$ do
      $L''.count \leftarrow L''.count - 1$
 for all literals $L'$ s.t. $P \cup \{L\} \models \neg L'$ do
   if ($L'.value = u$) then
    $L'.value \leftarrow f$
    $L'.level \leftarrow level$
    for all clauses $C$ with $L' \in C$ do
     $C.count \leftarrow C.count - 1$
```

```
backtrack($level$)
 for all literals $L$ do
  if ($L.level > level$) then
   if ($L.value = t$) then
    for all clauses $C$ s.t. $L \in C$
     if ($C.level > level$) then
      $C.sat \leftarrow f$
      $C.level \leftarrow -1$
      for all literals $L' \in C$ do
       $L'.count \leftarrow L'.count + 1$
   if ($L.value = f$) then
    for all clauses $C$ s.t. $L \in C$ then
     $C.count \leftarrow C.count + 1$
   $L.value \leftarrow u$
   $L.level \leftarrow -1$
```

Solve() starts by initializing the data structures and then makes the call **DPLL()**, which will return a partial interpretation P such that $P \models T$, if one exists; otherwise $false$ is returned.

Below are descriptions of the auxiliary functions used by **DPLL()**.

pickLiteral(): This function returns an unassigned literal with a maximal count of occurrences in not yet satisfied clauses. That is, we select from among literals L s.t. $L.value = u$ so as to maximize $L.count$. Of course, there are other possible heuristics, and in our experiments we tried just a few of them.

propagate($L, level$)**:** This function satisfies all literals L' s.t. $P \cup \{L\} \models L'$ and $P \not\models L'$. The function also satisfies all not yet satisfied clauses C containing such L' and decrements the count associated with each literal in each such C by 1. This function falsifies all unassigned literals L' s.t. $P \cup \{L\} \dashv L'$ and decrements $C.count$ of all clauses C s.t. $L' \in C$.

backtrack($level$)**:** This function restores the state just before the decision that began level $level + 1$.

The work done in propagation and backtracking in algorithm 6 is more compared to algorithm 4. During propagation and backtracking algorithm 6 has an extra overhead compared to algorithm 4 for keeping track of the current counts of occurrences in unsatisfied clauses.

Let's look at few bookkeeping heuristics now.

4.1.1 Bookkeeping Heuristic 1

This heuristic selects a positive literal which has maximum value of (*satisfaction* + *falsification*) where *satisfaction* is the number of clauses that would be satisfied if the positive literal is satisfied with respect to the current partial interpretation denoted by #s, and *falsification* is the number of literal occurences that would be falsified in clauses neither satisfied nor falsified with respect to the current partial interpretation, denoted by #f.

Let's look at an example of Heuristic 1.

Consider a theory T with variables $V = \{1, 2, 3\}$ such that for each $v \in V$, $dom(v) =$

$\{0,1\}$. The theory T consists of the three clauses given below.

$Clause1$ $\{1=0,2=0\}\,U(2)$

$Clause2$ $\{1=0,2=0\}\,U(2)$

$Clause3$ $\{1\neq 0,3\neq 0\}\,U(2)$

The heuristic will pick $1=0$ with value 2+1 = 3.

4.1.2 Bookkeeping Heuristic 2

This heuristic selects a positive literal which has maximum value of (*satisfaction* - *falsification*).

Let's look at an example of Heuristic 2.

Consider a theory T with variables $V=\{1,2,3\}$ such that for each $v\in V$, $dom(v)=\{0,1\}$. The theory T consists of the three clauses given below.

$Clause1$ $\{1=0,2=0\}\,U(2)$

$Clause2$ $\{1=0,2=0\}\,U(2)$

$Clause3$ $\{1\neq 0,3\neq 0\}\,U(2)$

The heuristic will pick $2=0$ with value 2-0 = 2.

4.1.3 Bookkeeping Heuristic 3

This heuristic selects a positive literal which has maximum value of (*satisfaction* + *units*) where *units* (a lower bound on) is the number of binary clauses that would become unit if the positive literal is satisfied with respect to the current partial interpretation, denoted by #u.

Let's look at an example of Heuristic 3.

Consider a theory T with variables $V=\{1,2,3\}$ such that for each $v\in V$, $dom(v)=\{0,1\}$. The theory T consists of the three clauses given below.

$Clause1\ \{1 = 0, 2 = 0\}\, U(2)$

$Clause2\ \{1 = 0, 2 = 0\}\, U(2)$

$Clause3\ \{1 \neq 0, 3 \neq 0\}\, U(2)$

The heuristic will pick $1 = 0$ with value 2+1 = 3.

4.2 Comparisons and Conclusions

The adoption of different heuristics can greatly affect the performance of the solver. We ran a set of comparisons designed to begin to address this question. We compared the performance of the VSIDS heuristic with the three other bookkeeping heuristics described above. From the comparisons we found that VSIDS performs better almost every time compared to all three bookkeeping heuristics.

It appears that the crucial thing about VSIDS is that the initial statistics are good and later there is bias towards the literals that occur in the most recent learned clauses. Bookkeeping heuristics that use current statistics plus some weighing function for learned clauses that could bias the search in favor of recently learned clauses could possibly help in improving the performance of solver.

A note about methodology. In general, SAT solvers are known to perform better on average if their implementation includes some nondeterministic elements. This is because solution times on hard problems are long-tailed (that is, if you vary the search on a given "hard" problem, there will be some solution attempts that take much longer, intuitively because the solver gets bogged down in a "bad" part of the earch space.). As a step in this direction we reinstated the nondeterministic tie-breaking element of VSIDS and similarly broke ties nondeterministically in the other heuristics as well. Then, to compare the performance of the different heuristics with this minimal level of nondeterminism, we used the mean and median of 10 test runs.

Chapter 5

More on Clause Learning and Non-Chronological Backtracking and Algorithm 7

The basic idea behind clause learning is the addition of a clause to a theory. The learned clause is formed taking into account the assignments due to which the theory is falsified with respect to the current partial interpretation, representing a constraint preventing the same assignments to occur in the future by generating unit literals for propagation before the theory is falsified (in that way) again. To understand the clause learning and non-chronological backtracking used by our implementation of Algorithm 4, additional relevant information is discussed in this chapter.

We use resolution to obtain the learned clause. The basic idea of resolution is to take two clauses and merge them to form a new clause while removing a literal and its complement. Consider two clauses, $C_1 = C \cup \{v = x\}$ and $C_2 = C' \cup \{v \neq x\}$. Resolution performed on C_1 and C_2 results in $C \cup C'$. Now, consider an interpretation I that satisfies both C_1 and C_2. Of course I must falsify one of $v = x$, $v \neq x$. If I falsifies $v = x$, it must satisfy C, since

I satisfies C_1. On the other hand, if I falsifies $v \neq x$, it must satisfy C', since I satisfies C_2. Either way, I satisfies $C \cup C'$. The above discussion shows that the original clauses C_1 and C_2 "entail" the *resolvent* (the clause obtained by resolving C_1 and C_2). That is, any interpretation that satisfies the original clauses also satisfies the resolvent. Therefore adding the resolvent to a theory containing C_1 and C_2 does not affect the models of the theory. Thus resolution is a sound inference method, and we can safely use it to help guide our search for a model.

The version of resolution that we use in our algorithm is slightly stronger and slightly more general. Notice that our prior discussion of soundness of resolution involved considering cases based on whether a certain literal is satisfied by a given interpretation. In our function, an additional parameter L plays the role of that literal. The clause that is returned by our function consists of (1) the literals from clause C that are satisfied by at least one interpretation that *does not satisfy* L, and (2) the literals from C' that are satisfied by at least one interpretation that *also satisfies* L. That is, **resolve**(C, L, C') = $\{L' \in C \mid \{L\} \not\models L'\} \cup \{L' \in C' \mid \{L\} \not\models\!\!| \, L'\}$.

Let's look at two examples of how **resolve**(C, L, C') works:

Example 1

Let us consider two clauses made up of variables $V = \{1,2,3\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$.

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

$L: \ 3 = 2$

$C': \ \{1 = 1, 2 = 0, 3 \neq 2\}$

$R: \ \{1 = 0, 2 = 2, 1 = 1, 2 = 0\}$ is the clause returned by the function **resolve(**C, L, C'**).**

Example 2

Let us consider two clauses made up of variables $V = \{1,2,3\}$ such that for each $v \in V$, $dom(v) = \{0,1,2\}$.

$C: \{1=0, 2=2, 3=2\}$

$L: 3=2$

$C': \{1=1, 2=0, 3=0, 3=1\}$

$R: \{1=0, 2=2, 1=1, 2=0\}$ is the clause returned by the function **resolve(**C, L, C'**).**

To show that the definition of **resolve**(C, L, C') is sound, we need to show that every model of $\{C, C'\}$ satisfies **resolve**(C, L, C').

Assume interpretation I satisfies $\{C, C'\}$. We need to show I satisfies **resolve**(C, L, C').

Consider two cases.

Case 1: $I \models L$. Since $I \models C'$, I satisfies one of the literals in C' that can be satisfied by some interpretation that also satisfies L. Consequently, I satisfies **resolve**(C, L, C').

Case 2: Otherwise. Since $I \models C$, I satisfies one of the literals in C that can be satisfied by some interpretation that does not satisfy L. Consequently, I satisfies **resolve**(C, L, C').

So, when some clause $C \in T$ is falsified w.r.t. P, we use resolution to form the learned clause. For this purpose, we define a "reason" for each literal L that is falsified by the current partial interpretation P. This "reason" is a specific clause, *L.reason*, obtained in one of the following three ways. If L became false because of a unit literal L' in P s.t. $\{L'\} \rightleftharpoons\!\!| L$, then *L.reason* is the unit clause that produced unit literal L'. If L became false because of a decision literal L' in P s.t. $\{L'\} \rightleftharpoons\!\!| L$, then *L.reason* is the clause $\{L', \bar{L'}\}$. (Notice that the clause $\{L', \bar{L'}\}$ is saisfied by every interpretation, and so by every model of T.) Otherwise *L.reason* is the clause $\{v = x \mid v$ is the variable that occurs in L and $x \in dom(v)\}$. (Again, such a clause is satisfied by every interpretation.)

We should observe that of these three cases for *L.reason*, only the first is needed for clause learning in the Boolean setting. The need for the third case in the Finite Domain setting was recognized in Lal's thesis. Unfortunately, as we will discuss shortly, Lal's thesis did not recognize the need for the second case, leading to a subtle error in his clause learning algorithm.

So, in the clause learning algorithm, we take the reason clause, *L.reason*, of the literal *L* falsified latest in conflict clause *C*. We resolve clause *C* and *L.reason* w.r.t. to literal *L*. We check to see if the resultant clause has only one literal that was assigned a value at the current level (that way, we can backtrack so as to make the resolvent a unit clause, and continue the search from there!). If so, then we are done. If not, the process is repeated till we get a resolvent with only one literal that was falsified at the current level. The resolvent is then added to the theory and we backtrack to the appropriate level based on the learned clause. That is, we backtrack to the earliest level at which the learned clause is unit. After backtracking, the learned clause becomes unit and the search continues from there.

In Algorithm 7, there are global variables *T* and *level* for storing the theory and the current decision level. In addition, there is another global variable *index* for keeping track of the order in which literals are falsified.

Each literal *L* has four associated fields: *value*, *level*, *reason*, and *index*. Fields *L.value* and *L.level* work as before. *L.reason* is as described in previous discussion. *L.index* is used to keep track of the order in which literals were falsified in the current partial interpretation.

Algorithm 7

```
Solve()
 $level \leftarrow 0$
 $index \leftarrow 0$
 for all literals $L$ do
  $L.value \leftarrow u$
  $L.level \leftarrow -1$
 for all clauses $C$ do
  $C.sat \leftarrow f$
  $C.count \leftarrow$ number of literals in $C$
  $C.level \leftarrow -1$
 DPLL()
```

```
$T$ is a theory
**DPLL**()
 **while**($true$)
  **if** (**checkSat**() = $t$) **then**
   **return** $\{L \mid L.value = t\}$
  **if** (**checkSat**() = $f$) **then**
    **if** ($level = 0$) **then**
     **return** $false$
    **else**
     $C \leftarrow$ **conflictClause**()
     $learnedClause \leftarrow$ **analyzeConflict**($C$)
     $level \leftarrow$ **findBacktrackingLevel**($learnedClause$)
     $T \leftarrow T \cup \{learnedClause\}$
     **backtrack**($level$)
  $C \leftarrow$**findUnitClause**()
  **if** ($C \neq NULL$) **then**
   $L \leftarrow$**unitLiteral**($C$)
   $reason \leftarrow C$
  **else**
   $L \leftarrow$**pickLiteral**()
   $reason \leftarrow \{L, \bar{L}\}$
   $level \leftarrow level + 1$
  **propagate(**$L$, $reason$**)**
```

```
**conflictClause**()
 **for all** clauses $C \in T$ **do**
  **if (**$C.count = 0$**) then**
   **return** $C$
 **return** $NULL$
```

```
**findBacktrackingLevel**($C$)
 $max = -1$
 **for all** literals $L \in C$ **do**
  **if** ($L.level > max$ **and** $L.level < level$) **then**
   $max \leftarrow L.level$
 **if** ($max \neq -1$) **then**
  **return** $max$
 **else**
  **return** $level - 1$
```

```
**findUnitClause**()
 **for all** clauses $C \in T$ **do**
  **if** ($C.sat = f$ **and** $C.count = 1$) **then**
   **return** $C$
 **return** $NULL$
```

```
**unitLiteral**($C$)
 **return** a literal $L$ in $C$ s.t. $L.value = u$
```

```
**propagate**($L, reason$)
 $P \leftarrow \{L \mid L.value = t\}$
 $index \leftarrow index + 1$
 **for all** literals $L'$ s.t. $P \cup \{L\} \models L'$ **do**
  **if** ($L'.value = u$) **then**
   $L'.value \leftarrow t$
   $L'.level \leftarrow level$
   **for all** clauses $C$ s.t. $L' \in C$ **do**
    **if** ($C.sat = f$) **then**
     $C.sat \leftarrow t$
     $C.level \leftarrow level$
 **for all** literals $L'$ s.t. $P \cup \{L\} \Dashv L'$ **do**
   **if** ($L'.value = u$) **then**
    $L'.value \leftarrow f$
    $L'.level \leftarrow level$
    $L'.index \leftarrow index$
    **if** ($\{L\} \Dashv L'$) **then**
     $L'.reason \leftarrow reason$
    **else**
     $L'.reason \leftarrow \{v = x \mid v$ is the variable that occurs in $L'$ **and** $x \in dom(v)\}$
    **for all** clauses $C$ s.t. $L' \in C$ **do**
     $C.count \leftarrow C.count - 1$
```

```
backtrack(level)
 for all literals $L$ do
  if ($L.level > level$) then
   if ($L.value = t$) then
    for all clauses $C$ s.t. $L \in C$
     if ($C.level > level$) then
      $C.sat \leftarrow f$
      $C.level \leftarrow -1$
   if ($L.value = f$) then
    for all clauses $C$ s.t. $L \in C$ do
     $C.count \leftarrow C.count + 1$
   $L.value \leftarrow u$
   $L.level \leftarrow -1$
```

```
analyzeConflict(C)
 if (potent(C)) then
  return C
 $L \leftarrow$ maxLit($C$)
 $resolvent \leftarrow$ resolve($C$, $L$, $L.reason$)
 return analyzeConflict($resolvent$)
```

```
potent(C, level)
 if there is exactly one $L \in C$ s.t. $L.level = level$ then
  return $t$
 else
  return $f$
```

```
maxLit(C)
 return a literal $L \in C$ with maximal $L.index$
```

```
resolve(C, L, C')
 return $\{L' \in C \mid \{L\} \not\models L'\} \cup \{L' \in C' \mid \{L\} \not\dashv L'\}$
```

The functions **checkSat**(), and **pickLiteral**() work the same as before. Let's see the other functions:

conflictClause(): This function returns a clause falsified w.r.t. to the current partial interpretation, if one exists; otherwise $NULL$.

findUnitClause(): This function returns a clause unit w.r.t. to the current partial interpretation, if one exists; otherwise $NULL$.

unitLiteral(C): This function returns a literal unassigned w.r.t. to the curent partial interpretation in the clause C.

propagate($L, reason$): This function satisfies all literals L' s.t. $P \cup \{L\} \models L'$ and $P \not\models L'$. The function also satisfies all not yet satisfied clauses C containing such L'. This function falsifies all unassigned literals L' s.t. $P \cup \{L\} \mathrel{=\!\!\mid} L'$ and decrements $C.count$ of all clauses C s.t. $L' \in C$. Now this function also sets the *reason* and *index* fields for the falsified literals.

backtrack($level$): This function restores the state just before the decision that began level $level + 1$.

analyzeConflict($C, level$): This function returns a clause to be added to the theory i.e. the learned clause. The clause returned should contain exactly one literal at the current $level$.

potent(C, $level$): This function returns $true$ if the clause C contains exactly on only one literal L having $L.level = level$, else returns $false$.

maxLit(C): This function returns a literal $L \in C$ that was falsified with respect to the current partial interpretation after all other literals in C. That is, it returns literal $L \in C$ with maximal $L.index$.

findBacktrackingLevel(C): This function returns the second highest level associated with a literal $L \in C$. If the clause C consists of only one literal, then it returns $level - 1$. There are other options available, such as we can always return $level - 1$.

resolve(C, L, C'): This function returns the clause $\{L' \in C \mid \{L\} \not\models L'\} \cup \{L' \in C' \mid \{L\} \not\mathrel{=\!\!\mid} L'\}$. We use the same form of resolution as used by Lal [15].

Now let's look at few examples to see how Algorithm 7 works:

Example 1

Lets reconsider Example 2 from Section 2.4. Theory T is made up of variables $V = \{1,2\}$ such that for each $v \in V$, $dom(v) = \{0,1\}$. The theory T consists of the four clauses given below. The current partial interpretation $P = \emptyset$. **Solve()** initializes the data structures and makes the first call **DPLL()**.

$P = \emptyset$

$Clause1\ \{1 = 0,\ 2 = 0\}\ \ U(2)$

$Clause2\ \{1 \neq 0,\ 2 = 0\}\ \ U(2)$

$Clause3\ \{1 = 0,\ 2 \neq 0\}\ \ U(2)$

$Clause4\ \{1 \neq 0,\ 2 \neq 0\}\ \ U(2)$

T is neither satisfied nor falsified and there are no unit literals, so we pick a literal. Assume the "decision literal" is $1 = 0$ from $Clause1$. The *level* parameter is incremented to 1, since $1 = 0$ will be our first decision literal, and literal $1 = 0$ is propagated. As $1 = 0$ is a decision literal, the newly falsified literals $1 \neq 0$ and $1 = 1$ have their *reason* set to be the clause $\{1 = 0,\ 1 \neq 0\}$ and their *index* set to 1.

$P = \{1 = 0_1\}$

$Clause1\ \{\mathit{1} = \mathit{0},\ 2 = 0\}\ \ S(2)$

$Clause2\ \{$~~$1 \neq 0$~~$,\ 2 = 0\}\ \ U(1)$

$Clause3\ \{\mathit{1} = \mathit{0},\ 2 \neq 0\}\ \ S(2)$

$Clause4\ \{$~~$1 \neq 0$~~$,\ 2 \neq 0\}\ \ U(1)$

Now $Clause1.sat = t$, $Clause1.level = 1$ and $Clause3.sat = t$, $Clause3.level = 1$. Notice that $Clause2$ is unit, as $Clause2.sat = f$ and $Clause2.count = 1$. So we must satisfy unit literal $2 = 0$. So literal $2 = 0$ is propagated at level 1. As $2 = 0$ is a unit literal, the newly falsified literals $2 \neq 0$ and $2 = 1$ have their *reason* set to be $Clause2$ and their *index* set to be 2.

$P = \{1 = 0_1,\ 2 = 0_1\}$

$Clause1$ $\{\mathit{1 = 0},\ \mathit{2 = 0}\}$ $S(2)$

$Clause2$ $\{\cancel{1 \neq 0},\ \mathit{2 = 0}\}$ $S(1)$

$Clause3$ $\{\mathit{1 = 0},\ \cancel{2 \neq 0}\}$ $S(1)$

$Clause4$ $\{\cancel{1 \neq 0},\ \cancel{2 \neq 0}\}$ $F(0)$

Now $Clause2.sat = t$ and $Clause2.level = 1$. But T is falsified, as $Clause4.count = 0$. As current level is not equal to 0, $Clause5$ is learned and added to the theory. $Clause5$ is formed using the $Clause4$ and the reason clause of the literal falsified last in $Clause2$. As literal $2 \neq 0$ has index greater than index of $1 \neq 0$ in $Clause4$, we pick the reason clause of $2 \neq 0$ and $Clause4$ to form the learned $Clause5$. The function **resolve**($Clause2$, $2 \neq 0$, $Clause4$) will create $Clause5$ to be $\{1 \neq 0\}$ having only one literal $1 \neq 0$ at the current $level$. After learning the clause, we backtrack to level 0 based on learned $Clause5$.

$P = \emptyset$

$Clause1$ $\{1 = 0,\ 2 = 0\}$ $U(2)$

$Clause2$ $\{1 \neq 0,\ 2 = 0\}$ $U(2)$

$Clause3$ $\{1 = 0,\ 2 \neq 0\}$ $U(2)$

$Clause4$ $\{1 \neq 0,\ 2 \neq 0\}$ $U(2)$

$Clause5$ $\{1 \neq 0\}$ $U(1)$

After backtracking to level 0, $Clause5$ is unit, as $Clause5.sat = f$ and $Clause5.count = 1$. So we must satisfy unit literal $1 \neq 0$ in $Clause5$. Thus, literal $1 \neq 0$ is propagated at level 0. As $1 \neq 0$ is a unit literal, the newly falsified literal $1 = 0$ has its *reason* set to be $Clause5$ and *index* set to be 3.

$P = \{1 \neq 0_0\}$

$Clause1$ $\{\cancel{1 = 0},\ 2 = 0\}$ $U(1)$

$Clause2$ $\{\mathit{1 \neq 0},\ 2 = 0\}$ $S(2)$

$Clause3$ $\{\cancel{1 = 0},\ 2 \neq 0\}$ $U(1)$

$Clause4$ $\{\mathit{1 \neq 0},\ 2 \neq 0\}$ $S(2)$

$Clause5\ \{1 \neq 0\}$ $\qquad S(1)$

Now $Clause2.sat = t$, $Clause2.level = 0$ and $Clause4.sat = t$, $Clause4.level = 0$. Notice that $Clause1$ is unit, as $Clause1.sat = f$ and $Clause1.count = 1$. So we must satisfy unit literal $2 = 0$. So literal $2 = 0$ is propagated at level 0. As $2 = 0$ is a unit literal, the newly falsified literals $2 \neq 0$ and $2 = 1$ have their *reason* set to be $Clause1$ and *index* set to 4.

$P = \{1 \neq 0_0,\ 2 = 0_0\}$

$Clause1\ \{\cancel{1 = 0},\ 2 = 0\}$ $\quad S(1)$

$Clause2\ \{1 \neq 0,\ 2 = 0\}$ $\quad S(2)$

$Clause3\ \{\cancel{1 = 0},\ \cancel{2 \neq 0}\}$ $\quad F(0)$

$Clause4\ \{1 \neq 0,\ \cancel{2 \neq 0}\}$ $\quad S(1)$

$Clause5\ \{1 \neq 0\}$ $\qquad S(1)$

Now $Clause1.sat = t$ and $Clause1.level = 0$. But T is falsified, as $Clause3.count = 0$. As the theory is falsified with respect to the current partial interpretation when level equals 0, we return *false*, implying that the theory is unsatisfiable.

Example 2

Given below is a theory T with variables $V = \{1,2\}$ such that for each $v \in V$, $dom(v) = \{0,1\}$. The theory T consists of the two clauses given below. **Solve()** initializes the data structures and makes the first call **DPLL()**.

$P = \emptyset$

$Clause1\ \{1 = 0,\ 1 \neq 1\}\, U(2)$

$Clause2\ \{1 = 1,\ 1 \neq 0\}\, U(2)$

T is neither satisfied nor falsified and there are no unit literals, so a decision literal is picked. Assume the "decision literal" is $1 = 0$ from $Clause1$. The *level* parameter is incremented to 1, since $1 = 0$ will be our first decision literal, and literal $1 = 0$ is propagated. As $1 = 0$ is a decision literal, the literals newly falsified $1 \neq 0$ and $1 = 1$ have their *reason* set to be the clause $\{1 = 0,\ 1 \neq 0\}$ and *index* set to be 1.

$P = \{1 = 0_1\}$

$Clause1$ $\{$*1=0*, *1* $\neq$ *1*$\}$ $S(2)$

$Clause2$ $\{$~~$1 = 1$~~, ~~$1 \neq 0$~~$\}$ $F(0)$

Now $Clause1.sat = t$ and $Clause1.level = 1$. But T is falsified, as $Clause2.count = 0$. As current level is not equal to 0, $Clause3$ is learned and added to the theory. $Clause3$ is formed using the $Clause2$ and reason clause of the literal falsified last in $Clause2$. As literals $1 = 1$ and $1 \neq 0$ were falsified w.r.t. P together and so have the same index, we pick the reason clause of one of them to form the learned $Clause5$. Assume we pick the reason clause of $1 = 1$. The function **resolve**($Clause2$, $1 = 1$, $\{1 = 0,\ 1 \neq 0\}$) will create $Clause3$ to be $\{1 \neq 0\}$ having only one literal $1 \neq 0$ at the current *level*. After learning the clause we backtrack to level 0 based on learned $Clause3$.

$P = \emptyset$

$Clause1$ $\{1 = 0,\ 1 \neq 1\}$ $U(2)$

$Clause2$ $\{1 = 1,\ 1 \neq 0\}$ $U(2)$

$Clause3$ $\{1 \neq 0\}$ $U(1)$

After backtracking to level 0, $Clause3$ is unit, as $Clause3.sat = f$ and $Clause3.count = 1$. So we must satisfy unit literal $1 \neq 0$ in $Clause3$. Thus, literal $1 \neq 0$ is propagated at level 0. As $1 \neq 0$ is a unit literal, the newly falsified literal $1 = 0$ has its *reason* set to be $Clause3$ and *index* set to be 2.

$P = \{1 \neq 0_1\}$

$Clause1$ $\{$~~$1 = 0$~~, ~~$1 \neq 1$~~$\}$ $F(0)$

$Clause2$ $\{$*1 = 1*, *1* $\neq$ *0*$\}$ $S(2)$

$Clause3$ $\{$*1* $\neq$ *0*$\}$ $S(1)$

Now $Clause2.sat = t$ and $Clause2.level = 1$. But T is falsified, as $Clause1.count = 0$. As the theory is falsified with respect to the current partial interpretation when level equals

0, we return *false*, implying that the theory is unsatisfiable.

Note: The algorithm implemented by Lal [15] is essentially the same as Algorithm 7 but the L'*.reason* when L' is falsified because of a decision literal L was not set properly to be the clause $\{L, \bar{L}\}$. Consequenstly, Lal's version of the **analyzeConflict**() function fails to return a learned clause on problems such as that in Example 2 above. Due to this reason, Lal's solver crashes on certain inputs.

Chapter 6

Benchmark Problems and Experimental Results

To test our FD SAT solvers we used a random FD SAT problem generator developed by Lal [15]. Also we have tested our solver on structured benchmark problems GT_N [5, 7, 14], Pigeonhole [8] and Pebbling [4, 5, 6]. GT_N, Pigeonhole, and Pebbling are all families of unsatisfiable problem instances.

Benchmark Problems

6.1 GT_N

GT_N problems are unsatisfiable theories based on the observation that every partial order on a set $\{1,2,...,N\}$ must have a maximal element. So, a GT_N problem is satisfiable iff some partial order on some N element set has no maximal element. (Of course this is false, so each GT_N problem is unsatisfiable.) These problems are interesting because in practice they seem to be hard – essentially the solver must verify that every partial order of the set $\{1,2,...,N\}$ has a maximal element!

Variables of a GT_N theory are $x_{i,j}$ such that $i, j \in \{1,2,...,N\}$ and $i \neq j$. These are Boolean variables having domain $\{t, f\}$ or $\{0,1\}$ and, intuitively, $x_{i,j}$ encodes $i \prec j$. Clauses $\{x_{i,j} = 0,\ x_{j,i} = 0\}$, with $i, j \in \{1,...,N\}$ and $i < j$, ensure anti-symmetry, and clauses $\{x_{i,j} = 0,\ x_{j,k} = 0,\ x_{i,k} = 1\}$, with $i, j, k \in \{1,...,N\}$ and i, j, k all distinct, ensure transitivity on $i \prec j$. Anti-symmetry and transitivity make the relation a partial order on $\{1,2,...,N\}$. *Successor clauses* $\{x_{k,j} = 1 \mid k \neq j \textit{ and } j \in \{1,...,N\}\}$ where $k \in \{1,...,N\}$ provide a contradiction by saying that every element k has a successor. This is false for maximal elements of $\{1,2,...,N\}$ under $\prec$.

We show the $GT_{N=3}$ instance below. There are 6 Boolean variables $x_{i,j}$ such that $i \neq j$ and $i, j \in \{1,2,3\}$. In the clauses given below, the first three clauses ensure anti-symmetry, the next six ensure transitivity and the last three are successor clauses providing contradiction.

$x_{1,2} = 0$ $x_{2,1} = 0$

$x_{1,3} = 0$ $x_{3,1} = 0$

$x_{2,3} = 0$ $x_{3,2} = 0$

$x_{1,2} = 0$ $x_{2,3} = 0$ $x_{1,3} = 1$

$x_{1,3} = 0$ $x_{3,2} = 0$ $x_{1,2} = 1$

$x_{2,1} = 0$ $x_{1,3} = 0$ $x_{2,3} = 1$

$x_{2,3} = 0$ $x_{3,1} = 0$ $x_{2,1} = 1$

$x_{3,1} = 0$ $x_{1,2} = 0$ $x_{3,2} = 1$

$x_{3,2} = 0$ $x_{2,1} = 0$ $x_{3,1} = 1$

$x_{1,2} = 1$ $x_{1,3} = 1$

$x_{2,1} = 1$ $x_{2,3} = 1$

$x_{3,1} = 1$ $x_{3,2} = 1$

6.2 Pigeonhole

Pigeonhole problems are unsatisfiable theories based on the Pigeonhole property: you can't fit $n+1$ pigeons in n holes without at least two pigeons occupying the same hole. The formulation of the pigeon hole problem as a FD SAT problem is quite simple: $n+1$ variables are used to represent the $n+1$ pigeons, and each variable has the domain $\{0,...,n-1\}$, representing the n holes. Thus, the literal $i = j$ represents that pigeon i is placed in hole j ($i \in \{1,2,...,n+1\}$, $j \in \{0,1,...,n-1\}$). The finite domain theory consists of $n^2(n+1)/2$ clauses $\{i \neq k, j \neq k\}$ representing that pigeons i and j are not placed in the same hole k, where $i < j$.

An example Pigeonhole Problem for fitting 4 pigeons in 3 holes is given below.

$1 \neq 0$ $2 \neq 0$

$1 \neq 0$ $3 \neq 0$

$1 \neq 0$ $4 \neq 0$

$2 \neq 0$ $3 \neq 0$

$2 \neq 0$ $4 \neq 0$

$3 \neq 0$ $4 \neq 0$

$1 \neq 1$ $2 \neq 1$

$1 \neq 1$ $3 \neq 1$

$1 \neq 1$ $4 \neq 1$

$2 \neq 1$ $3 \neq 1$

$2 \neq 1$ $4 \neq 1$

$3 \neq 1$ $4 \neq 1$

$1 \neq 2$ $2 \neq 2$

$1 \neq 2$ $3 \neq 2$

$1 \neq 2$ $4 \neq 2$

$2 \neq 2$ $3 \neq 2$

$2 \neq 2$ $4 \neq 2$

$3 \neq 2$ $4 \neq 2$

6.3 Pebbling

A *pebbling graph* is a finite acyclic directed graph whose nodes are clauses – with no variables in common between clauses – such that each node has indegree 0 or 2 and outdegree 0 or 1. A *pebbling* of a pebbling graph G is an interpretation I s.t. for all nodes C of G, if I satisfies every predecessor of C, then $I \models C$.

For a pebbling graph G, the corresponding *pebbling problem* is constructed from G as follows. Nodes with indegree 0 are called *source nodes*. Nodes with outdegree 0 are called *target nodes*. We construct a theory that will be satisfied only by a pebbling that satisfies no target nodes. Since no such pebbling exists, the theory will be unsatisfiable (and in general this will be "hard" to prove). More precisely,

- For every source node C, we include the clause C.
- For every target node $\{L_1, ..., L_n\}$ we include the clauses $\{\bar{L}_1\}$, ..., $\{\bar{L}_n\}$.
- For every node C of indegree 2, where the predecessors of C are $\{L_1, ..., L_m\}$ and $\{L'_1, ..., L'_n\}$, we include all clauses $\{\bar{L_i}, \bar{L'_j}\} \cup C$, where $i \in \{1, ..., m\}$ and $j \in \{1, ..., n\}$.

The pebbling problem given below is constructed from a pebbling graph having three nodes. Two source nodes and one target node. The source nodes contribute the first two clauses. The target node $\{3 = 1, 4 = 1\}$ accounts for the next two clauses. The remaining clauses are included owing to the target node with indegree 2 having the source nodes as predecessors.

$1 = 0\ 1 = 1\ 1 = 2$

$2 = 0\ 2 = 1\ 2 = 2$

$3 \neq 1$

$4 \neq 1$

$1 \neq 0\ 2 \neq 0\ 3 = 1\ 4 = 1$

$1 \neq 0\ 2 \neq 1\ 3 = 1\ 4 = 1$

$1 \neq 0\ 2 \neq 2\ 3 = 1\ 4 = 1$

$1 \neq 1\ 2 \neq 0\ 3 = 1\ 4 = 1$

$1 \neq 1\ 2 \neq 1\ 3 = 1\ 4 = 1$

$1 \neq 1\ 2 \neq 2\ 3 = 1\ 4 = 1$

$1 \neq 2\ 2 \neq 0\ 3 = 1\ 4 = 1$

$1 \neq 2\ 2 \neq 1\ 3 = 1\ 4 = 1$

$1 \neq 2\ 2 \neq 2\ 3 = 1\ 4 = 1$

6.4 Random Problems

We use a random FD SAT problem generator developed by Lal [15] to generate random satisfiable problems. The generator generates FD SAT problems based on input criteria such as number of variables, size of domain, number of clauses, and size of clauses. To generate a satisfiable problem the generator first creates a random interpretation using the input variables and domain. Then each generated clause includes one randomly literal satisfied by this interpretation and the rest of the clause is filled with other literals (positive/negative which are distinct from this literal) randomly. By following this method the generator guarantees to have at least one model of the problem.

Experimental Results

6.5 Comparison of FD-WL and FD-BK Solvers

These experiments are aimed at comparing the performance of the watched literals and the bookkeeping approach. We compare the search time, number of decisions, number of backtracks and the backtrack time on the same set of problems produced by the two solvers FD-WL and FD-BK. To compare the performance of the two solvers we used the same *VSIDS* heuristic introduced by the landmark Boolean SAT solver Chaff [18].

6.5.1 Comparison on Backtrack Time

Backtrack time is a crucial factor in determining the overall performance of the solvers, especially on hard problems where the time spent during backtracking may be significant. These set of experiments are aimed to answer the question whether FD-WL wins or loses compared to FD-BK while backtracking.

File Name	Backtrack Time	
	FD-WL	FD-BK
v6_c75	0	0
v7_c126	0	0
v8_c196	0	0
v9_c288	0	0.01
v10_c405	0	0.41
v12_c726	0.08	52.66
v13_c936	0.26	567.89

Table 1: Comparison of backtrack time taken by FD-BK and FD-WL on Pigeonhole problems.

File Name	Backtrack Time	
	FD-WL	FD-BK
v7_c252_d3_s0	0	0.01
v7_c1033_d4_s0	0	0.01
v7_c3134_d5_s0	0	0.05
v8_c739_d3_s0	0	0
v8_c4106_d4_s0	0	0.1
v8_c15635_d5_s0	0	9.43
v9_c2198_d3_s0	0	0.07
v9_c16395_d4_s0	0.03	12.3
v9_c78136_d5_s0	0.16	435.15
v10_c6573_d3_s0	0	0.7
v10_c65548_d4_s0	0.12	407.84

Table 2: Comparison of backtrack time taken by FD-BK and FD-WL on Pebbling problems.

File Name	Backtrack Time	
	FD-WL	FD-BK
v30_c141_d2_s0	0	0
v42_c238_d2_s0	0	0
v56_c372_d2_s0	0	0
v72_c549_d2_s0	0	0.01
v90_c775_d2_s0	0	0
v110_c1056_d2_s0	0	0
v156_c1807_d2_s0	0	0
v210_c2850_d2_s0	0	0.01
v272_c4233_d2_s0	0	0.03
v380_c7050_d2_s0	0	0.03

Table 3: Comparison of backtrack time taken by FD-BK and FD-WL on GT_N problems.

File Name	Backtrack Time	
	FD-WL	FD-BK
v100_c100_d3_s1	0	0
v100_c1000_d3_s1	0	0
v200_c200_d3_s1	0	0
v200_c2000_d3_s1	0	0
v300_c300_d3_s1	0	0
v300_c3000_d3_s1	0	0
v500_c500_d3_s1	0	0
v500_c5000_d3_s1	0	0

Table 4: Comparison of backtrack time taken by FD-BK and FD-WL on random satisfiable problems.

Looking at the backtrack time comparision of FD-WL and FD-BK in Table 1, Table 2, Table 3 and Table 4 we see that FD-WL almost always beats FD-BK while backtracking, as expected.

6.5.2 Comparison on Search Time, Number of Decisions, and Number of Backtracks

Here, we compare the performance of FD-WL and FD-BK. The search time, number of decisions of decisions and number of backtracks are crucial statistical indicators of how the solver performs.

File Name	Search Time		# of Decisions		# of Backtracks	
	FD-WL	FD-BK	FD-WL	FD-BK	FD-WL	FD-BK
v6_c75	0	0	49	49	49	49
v7_c126	0.02	0.01	129	129	129	129
v8_c196	0.08	0.07	321	321	321	321
v9_c288	0.42	0.35	769	769	769	769
v10_c405	2.12	1.9	1793	1793	1793	1793
v12_c726	117.12	149.2	9217	9217	9217	9217
v13_c936	992.58	1655.14	20481	20481	20481	20481

Table 5: Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FD-WL	FD-BK	FD-WL	FD-BK	FD-WL	FD-BK
v7_c252_d3_s0	0.01	0.01	80	80	80	80
v7_c1033_d4_s0	0.05	0.04	255	255	255	255
v7_c3134_d5_s0	0.37	0.27	624	624	624	624
v8_c739_d3_s0	0.05	0.03	242	242	242	242
v8_c4106_d4_s0	0.82	0.59	1023	1023	1023	1023
v8_c15635_d5_s0	24.59	28.93	3124	3124	3124	3124
v9_c2198_d3_s0	0.36	0.24	728	728	728	728
v9_c16395_d4_s0	37.68	39.32	4095	4095	4095	4095
v9_c78136_d5_s0	1394.65	1629.45	15624	15624	15624	15624
v10_c6573_d3_s0	4.45	2.8	2186	2186	2186	2186
v10_c65548_d4_s0	1123.38	1499.89	16383	16383	16383	16383

Table 6: Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FD-WL	FD-BK	FD-WL	FD-BK	FD-WL	FD-BK
v30_c141_d2_s0	0	0	10	10	10	10
v42_c238_d2_s0	0	0.01	15	15	15	15
v56_c372_d2_s0	0.01	0	21	21	21	21
v72_c549_d2_s0	0.01	0.01	28	28	28	28
v90_c775_d2_s0	0.03	0.02	36	36	36	36
v110_c1056_d2_s0	0.05	0.04	45	45	45	45
v156_c1807_d2_s0	0.14	0.1	66	66	66	66
v210_c2850_d2_s0	0.37	0.27	91	91	91	91
v272_c4233_d2_s0	0.86	0.64	120	120	120	120
v380_c7050_d2_s0	2.66	1.94	171	171	171	171

Table 7: Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on GT_N problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FD-WL	FD-BK	FD-WL	FD-BK	FD-WL	FD-BK
v100_c100_d3_s1	0	0.01	53	53	0	0
v100_c1000_d3_s1	0	0	8	8	0	0
v200_c200_d3_s1	0.01	0.01	149	149	0	0
v200_c2000_d3_s1	0.01	0	17	17	0	0
v300_c300_d3_s1	0.01	0.01	252	252	0	0
v300_c3000_d3_s1	0.01	0.01	24	24	0	0
v500_c500_d3_s1	0.02	0.05	384	384	0	0
v500_c5000_d3_s1	0.05	0.04	40	40	0	0

Table 8: Comparison of search time, number of decisions, number of backtracks done by FD-BK and FD-WL on random satisfiable problems.

On comparing the number of decisions and number of backtracking of the FD-WL and FD-BK in Tables 5, Table 6, Table 7, and Table 8 we observe that we get the same number of decisions in both the cases. Same number of both decisions and backtracks reflects the fact that both the solvers make the same decisions and create the same conflicts.

Looking at the search time comparision of FD-WL and FD-BK in Table 5, Table 6, Table 7, and Table 8 we see that FD-BK beats FD-WL on smaller sized problems and FD-

WL beats FD-BK on hard problems. Specifically, we can see that FD-WL beats FD-BK on the Pigeon Hole problems v12_c726 and v13_c936, Pebbling problems v8_c15635_d5_s0, v9_c16395_d4_s0, v9_c78136_d5_s0, and v10_c65548_d4_s0. We couldn't find GT_N problems where FD-WL beats FD-BK because the time required for backtracking on GT_N problems is very less. Also, the random satisfiable problems that we had were too easy, requiring no backtracks, so we do not report results for them in our comparison of FD-WL and FD-BK. FD-BK always wins when there is no backtracking.

6.6 Heuristic Comparisons

In Chapter 4 we learnt that heuristics based on statistics can be used by adding *L.count* to Algorithm 4. The set of experiments in this section aim to answer the question whether bookkeeping is improved by adding statistics based on *L.count* for picking decision literals or we can do better with a "weak" heuristic such as VSIDS. As described in Chapter 4, we experimented with three different heuristics using satistics based on *L.count*. The search time, number of decisions, and number of backtracks produced by each of the three heuristics will be compared with VSIDS on the same set of problems. The search time is probably the best indicator of the performance. The number of decisions is also used as a performance statistic with fewer decisions indicating smarter decisions. Of course, the number of decisions doesn't tell the real story as smaller number of decisions can still lead to more units and thereby more time spent in propagation. Similarly, smaller number of backtracks do not always indicate a good performance. But, a combination of search time, number of decisions and number of backtracks taken togather provides good statistical evidence of whether one heuristic is better than the other, which will help us answer the question whether heuristics based on *L.count* statistics win or lose over a heuristic such as VSIDS which has a relatively lower computational overhead. For comparing FD-BK and

FD-WL the algorithm was deterministic (we removed the randomness component present in VSIDS) as we were comparing the speed of the implementations. However, if we want to understand performance more realistically we need the algorithm to be non-deterministic, as discussed in Chapter 4. As we do want to compare the performance of different heuristics we include non-determinism in the algorithm for the following set of experiments and use the mean and median of 10 test runs to produce results on search time, number of decisions and number of backtracks for comparisons.

6.6.1 Bookkeeping Heuristic 1 vs. VSIDS

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v6_c75	0	0.01	53.9	93.5	49.8	89.9
v7_c126	0.02	0.04	148.4	261.6	132.4	257
v8_c196	0.11	0.14	360.9	656	323.8	650
v9_c288	0.52	0.68	886	1616.5	777.6	1609.2
v10_c405	3.26	3.65	2083.4	3828.8	1809.5	3819.8

Table 9: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v7_c252_d3_s0	0	0.01	88.5	120	80.1	120
v7_c1033_d4_s0	0.05	0.07	287.9	495	255.1	495
v7_c3134_d5_s0	0.46	0.5	728.1	1440	624.4	1440
v8_c739_d3_s0	0.04	0.05	263	363	242.3	363
v8_c4106_d4_s0	0.97	1.01	1159.8	2015	1023.5	2015
v8_c15635_d5_s0	29.39	51.71	3640	7447	3126.5	7447
v9_c2198_d3_s0	0.31	0.35	799.1	1092	729	1092
v9_c16395_d4_s0	40.27	71.94	4657.8	8127	4098.6	8127
v10_c6573_d3_s0	4.54	5.11	2392	3279	2189	3279

Table 10: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v30_c141_d2_s0	0	0.01	11.6	107.3	10.6	95.7
v42_c238_d2_s0	0.01	0.04	21.6	306.9	19.6	280.1
v56_c372_d2_s0	0.03	0.11	41.8	519.4	38.4	474.7
v72_c549_d2_s0	0.04	0.78	40.4	1846.1	35	1721.3
v90_c775_d2_s0	0.26	4.26	213.8	4907.7	144.4	4577.8
v110_c1056_d2_s0	0.55	48.81	307	13817.5	203	12957
v156_c1807_d2_s0	1.72	600*	686.8	-	400.4	-
v210_c2850_d2_s0	1.2	600*	278.4	-	169.6	-
v272_c4233_d2_s0	2.89	600*	556.8	-	262.6	-

Table 11: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs GT_N problems. (* = time out, 600 seconds)

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v100_c100_d3_cs3_s1	0	0.01	53.8	41.2	0	0.9
v100_c1000_d3_cs3_s1	0	0.02	7.8	26.4	0	20.4
v200_c200_d3_cs3_s1	0.01	0.03	144.8	78.3	0	0.8
v200_c2000_d3_cs3_s1	0.01	0.31	17.2	261	0	224.8
v300_c300_d3_cs3_s1	0.02	0.07	249.3	102.1	0	0
v300_c3000_d3_cs3_s1	0.02	1.01	24.1	523.1	0	441.5
v500_c500_d3_cs3_s1	0.05	0.18	384.7	177.5	0	0

Table 12: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on random satisfiable problems..

Table 9, Table 10, Table 11, and Table 12 all show that BK-V (i.e. VSIDS) outperforms BK-H1 (i.e. (#s+#f)) on mean search time over 10 runs on all three families of structured problems, as well as on random satisfiable problems. Also Table 9, Table 10, Table 11 show that BK-V produces lesser mean number of decisions and backtracks over 10 runs compared to BK-H1 on structured unsatisfiable problems. Table 12 shows that on small random satisfiable problems BK-V produces more mean decisions compared to BK-H1 over 10 runs but still has a better overall search time due to low computational cost. Table 12 shows that on some hard random satisfiable problems BK-V produces lesser mean decisions and no backtracks over 10 runs compared to BK-H1 and has a low overall search time.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v6_c75	0.01	0.01	54	93	49	89.5
v7_c126	0.02	0.04	148	257.5	131	254
v8_c196	0.10	0.14	359	660	322	654
v9_c288	0.49	0.68	887	1620	773	1612.5
v10_c405	3.01	3.65	2090	3831	1804	3822.5

Table 13: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v7_c252_d3_s0	0.01	0.01	88	120	80	120
v7_c1033_d4_s0	0.05	0.07	287	495	255	495
v7_c3134_d5_s0	0.38	0.5	724	1440	624	1440
v8_c739_d3_s0	0.04	0.05	262	363	242	363
v8_c4106_d4_s0	0.77	1	1149	2015	1023	2015
v8_c15635_d5_s0	29.41	51.38	3641	7447	3127	7447
v9_c2198_d3_s0	0.27	0.35	799	1092	729	1092
v9_c16395_d4_s0	40.32	71.95	4648	8127	4098	8127
v10_c6573_d3_s0	4.32	5.05	2388	3279	2188	3279

Table 14: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v30_c141_d2_s0	0	0.01	12	104	10	94
v42_c238_d2_s0	0.01	0.04	20	306.5	20	276.5
v56_c372_d2_s0	0.02	0.11	27	526	25	479
v72_c549_d2_s0	0.04	0.73	39	1794.5	32	1680
v90_c775_d2_s0	0.08	4.39	52	5027	43	4691
v110_c1056_d2_s0	0.24	54.16	158	14457.5	99	13597
v156_c1807_d2_s0	1.26	600*	565	-	348	-
v210_c2850_d2_s0	0.79	600*	135	-	121	-
v272_c4233_d2_s0	1.67	600*	179	-	147	-

Table 15: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on GT_N problems. (* = time out, 600 seconds)

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1	FDBK-V	FDBK-H1
v100_c100_d3_cs3_s1	0	0.01	53	42	0	1
v100_c1000_d3_cs3_s1	0	0.02	8	27	0	21
v200_c200_d3_cs3_s1	0.01	0.03	148	80	0	1
v200_c2000_d3_cs3_s1	0.01	0.32	17	268	0	227
v300_c300_d3_cs3_s1	0.02	0.07	248	102	0	0
v300_c3000_d3_cs3_s1	0.02	0.94	24	486	0	411.5
v500_c500_d3_cs3_s1	0.05	0.18	382	179	0	0

Table 16: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 1) over 10 runs on random satisfiable problems.

Table 13, Table 14, Table 15, and Table 16 show that BK-V outperforms BK-H1 on median search time over 10 runs on all the three families of structured problems as well as on random satisfiable problems. Also, Table 13, Table 14, Table 15 show that BK-VSIDS produces lesser number of median decisions and backtracks compared to BK-H1 over 10 runs on structured unsatisfiable problems. Table 16 shows that on small random satisfiable problems BK-V produces more median decisions compared to BK-H1 over 10 runs but still has a better overall search time due to low computational cost. Table 16 also shows that on hard random satisfiable problems BK-V produces lesser median number of decisions and no backtracks over 10 runs compared to BK-H1 and has a low overall search time.

6.6.2 Bookkeeping Heuristic 2 vs. VSIDS

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v6_c75	0	0	53.9	89.9	49.8	85.6
v7_c126	0.02	0.05	148.4	311.5	132.4	303.9
v8_c196	0.11	0.34	360.9	825.1	323.8	811.8
v9_c288	0.52	1.48	886	1525.7	777.6	1508.3
v10_c405	3.26	11.62	2083.4	3947	1809.5	3927.9

Table 17: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v7_c252_d3_s0	0	0.01	88.5	120	80.1	120
v7_c1033_d4_s0	0.05	0.09	287.9	495	255.1	495
v7_c3134_d5_s0	0.46	0.72	728.1	1440	624.4	1440
v8_c739_d3_s0	0.04	0.06	263	363	242.3	363
v8_c4106_d4_s0	0.97	1.5	1159.8	2015	1023.5	2015
v8_c15635_d5_s0	29.39	53.76	3640	7447	3126.5	7447
v9_c2198_d3_s0	0.31	0.52	799.1	1092	729	1092
v9_c16395_d4_s0	40.27	74.84	4657.8	8127	4098.6	8127
v10_c6573_d3_s0	4.54	6.82	2392	3279	2189	3279

Table 18: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v30_c141_d2_s0	0	0	11.6	10.8	10.6	10.8
v42_c238_d2_s0	0.01	0	21.6	19.2	19.6	17.8
v56_c372_d2_s0	0.03	0	41.8	27.9	38.4	26.2
v72_c549_d2_s0	0.04	0.01	40.4	50.5	35	40.9
v90_c775_d2_s0	0.26	0.03	213.8	76.3	144.4	56.2
v110_c1056_d2_s0	0.55	0.05	307	91.4	203	69
v156_c1807_d2_s0	1.72	0.13	686.8	125.6	400.4	101.2
v210_c2850_d2_s0	1.2	0.27	278.4	184.8	169.6	137.8
v272_c4233_d2_s0	2.89	0.57	556.8	288.6	262.6	190

Table 19: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on GT_N problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v100_c100_d3_cs3_s1	0	0.01	53.8	45.1	0	0
v100_c1000_d3_cs3_s1	0	0	7.8	12.1	0	0
v200_c200_d3_cs3_s1	0.01	0.03	144.8	103.8	0	0
v200_c2000_d3_cs3_s1	0.01	0.02	17.2	22.7	0	0
v300_c300_d3_cs3_s1	0.02	0.08	249.3	157.8	0	0
v300_c3000_d3_cs3_s1	0.02	0.05	24.1	40.2	0	0
v500_c500_d3_cs3_s1	0.05	0.22	384.7	244.1	0	0

Table 20: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on random satisfiable problems.

Table 17, Table 18 and Table 20 show that BK-V outperforms BK-H2 (i.e. (#s-#f)) on mean search time over 10 runs. Table 17 and Table 18 show that BK-V outperforms BK-H2 on mean number of decisions and backtracks. Table 19 shows that BK-H2 performs better than BK-V on GT_N problems on mean search time, number of decisions and number of backtracks over 10 runs. Table 20 shows that on random satisfiable problems BK-V produces more number of mean decisions over 10 runs comapred to BK-H2 on smaller problems and less number of mean decisions over 10 runs comapred to BK-H2 on harder problems. Even when BK-V produces more mean number of decisions compared to BK-H2 on smaller random satisfiable problems, BK-V outperforms BK-H2 on mean search time due to low computational cost of VSIDS compared to H2.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v6_c75	0.01	0	54	91	49	85.5
v7_c126	0.02	0.05	148	315	131	310.5
v8_c196	0.10	0.29	359	733	322	730.5
v9_c288	0.49	1.36	887	1368	773	1425
v10_c405	3.01	9.51	2090	3576	1804	3569

Table 21: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v7_c252_d3_s0	0.01	0.01	88	120	80	120
v7_c1033_d4_s0	0.05	0.09	287	495	255	495
v7_c3134_d5_s0	0.38	0.72	724	1440	624	1440
v8_c739_d3_s0	0.04	0.06	262	363	242	363
v8_c4106_d4_s0	0.77	1.49	1149	2015	1023	2015
v8_c15635_d5_s0	29.41	53.74	3641	7447	3127	7447
v9_c2198_d3_s0	0.27	0.52	799	1092	729	1092
v9_c16395_d4_s0	40.32	74.83	4648	8127	4098	8127
v10_c6573_d3_s0	4.32	6.73	2388	3279	2188	3279

Table 22: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v30_c141_d2_s0	0	0	12	11	10	11
v42_c238_d2_s0	0.01	0	20	18	20	18
v56_c372_d2_s0	0.02	0	27	26	25	26
v72_c549_d2_s0	0.04	0.01	39	35	32	34
v90_c775_d2_s0	0.08	0.02	52	48	43	45.5
v110_c1056_d2_s0	0.24	0.05	158	75	99	62.5
v156_c1807_d2_s0	1.26	0.12	565	113	348	96
v210_c2850_d2_s0	0.79	0.26	135	173	121	125.5
v272_c4233_d2_s0	1.67	0.46	179	210	147	166

Table 23: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on GT_N problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2	FDBK-V	FDBK-H2
v100_c100_d3_cs3_s1	0	0.01	53	45	0	0
v100_c1000_d3_cs3_s1	0	0	8	12	0	0
v200_c200_d3_cs3_s1	0.01	0.03	148	104	0	0
v200_c2000_d3_cs3_s1	0.01	0.02	17	23	0	0
v300_c300_d3_cs3_s1	0.02	0.08	248	158	0	0
v300_c3000_d3_cs3_s1	0.02	0.05	24	39	0	0
v500_c500_d3_cs3_s1	0.05	0.22	382	244	0	0

Table 24: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 2) over 10 runs on random satisfiable problems.

Table 21, Table 22 and Table 24 show that BK-V outperforms BK-H2 on median search time over 10 runs. Table 21 and Table 22 show that BK-V outperforms BK-H2 on median number of decisions and backtracks over 10 runs. Table 23 shows that BK-H2 performs better than BK-V on GT_N problems on median search time, number of decisions and number of backtracks over 10 runs. Table 24 shows that on random satisfiable problems BK-V produces more number of median decisions over 10 runs compared to BK-H2 on smaller problems and less number of median decisions over 10 runs compared to BK-H2 on harder problems. Even when BK-V produces more median number of decisions compared to BK-H2 on smaller random satisfiable problems, BK-V outperforms BK-H2 on median search time due to low computational cost of VSIDS compared to H2.

6.6.3 Bookkeeping Heuristic 3 vs. VSIDS

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v6_c75	0	0	53.9	98.3	49.8	95.5
v7_c126	0.02	0.04	148.4	266.7	132.4	263.2
v8_c196	0.11	0.26	360.9	671.5	323.8	667.8
v9_c288	0.52	1.48	886	1568.6	777.6	1562.3
v10_c405	3.26	9.21	2083.4	3688.6	1809.5	3680.9

Table 25: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v7_c252_d3_s0	0	0.01	88.5	116	80.1	116
v7_c1033_d4_s0	0.05	0.1	287.9	442.1	255.1	442.1
v7_c3134_d5_s0	0.46	0.73	728.1	1193.6	624.4	1193.6
v8_c739_d3_s0	0.04	0.07	263	358	242.3	358
v8_c4106_d4_s0	0.97	1.67	1159.8	1881	1023.5	1881
v8_c15635_d5_s0	29.39	56.96	3640	6521.6	3126.5	6521.6
v9_c2198_d3_s0	0.31	0.62	799.1	1086	729	1086
v9_c16395_d4_s0	40.27	84.11	4657.8	7800.2	4098.6	7800.2
v10_c6573_d3_s0	4.54	7.85	2392	3272	2189	3272

Table 26: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v30_c141_d2_s0	0	0	11.6	14	10.6	13.9
v42_c238_d2_s0	0.01	0	21.6	23	19.6	22
v56_c372_d2_s0	0.03	0.01	41.8	44	38.4	37.9
v72_c549_d2_s0	0.04	0.03	40.4	95.9	35	79.8
v90_c775_d2_s0	0.26	0.05	213.8	114.3	144.4	87.5
v110_c1056_d2_s0	0.55	0.15	307	273.3	203	178.7
v156_c1807_d2_s0	1.72	0.62	686.8	607.5	400.4	400.4
v210_c2850_d2_s0	1.2	1.32	278.4	1219.2	169.6	536.8
v272_c4233_d2_s0	2.89	5.25	556.8	2846.4	262.6	1196.6

Table 27: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on GT_N problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v100_c100_d3_cs3_s1	0	0	53.8	34.2	0	0
v100_c1000_d3_cs3_s1	0	0	7.8	5	0	0
v200_c200_d3_cs3_s1	0.01	0.03	144.8	75.6	0	0
v200_c2000_d3_cs3_s1	0.01	0.02	17.2	10.9	0	0
v300_c300_d3_cs3_s1	0.02	0.06	249.3	111.1	0	0
v300_c3000_d3_cs3_s1	0.02	0.04	24.1	15	0	0
v500_c500_d3_cs3_s1	0.05	0.19	384.7	191.5	0	0

Table 28: Comparison of mean of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on random satisfiable problems.

Table 25, Table 26, Table 27 show that BK-V outperforms BK-H3 (i.e. (#s+#u)) on mean search time, number of decisions and number of backtracks over 10 runs indicating that VSIDS is a better than Heuristic 3. Table 28 shows displays that BK-V outperforms BK-H3 i.e. (#s+#u) on mean search time over 10 runs. Table 28 also shows that on random satisfiable problems BK-V produces more number of mean decisions over 10 runs compared to BK-H3. Even when BK-V produces more mean number of decisions compared to

BK-H3, BK-V outperforms BK-H3 on mean search time due to low computational cost of VSIDS compared to H3.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v6_c75	0.01	0.01	54	96.5	49	93
v7_c126	0.02	0.04	148	262.5	131	261
v8_c196	0.10	0.25	359	661.5	322	657
v9_c288	0.49	1.42	887	1540.5	773	1533.5
v10_c405	3.01	8.99	2090	3664.5	1804	3656.5

Table 29: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pigeonhole problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v7_c252_d3_s0	0.01	0.01	88	116	80	116
v7_c1033_d4_s0	0.05	0.1	287	442	255	442
v7_c3134_d5_s0	0.38	0.73	724	1193	624	1193
v8_c739_d3_s0	0.04	0.07	262	358	242	358
v8_c4106_d4_s0	0.77	1.67	1149	1881	1023	1881
v8_c15635_d5_s0	29.41	56.93	3641	6521.5	3127	6521.5
v9_c2198_d3_s0	0.27	0.62	799	1086	729	1086
v9_c16395_d4_s0	40.32	84.05	4648	7800	4098	7800
v10_c6573_d3_s0	4.32	7.71	2388	3272	2188	3272

Table 30: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on Pebbling problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v30_c141_d2_s0	0	0	12	11.5	10	11.5
v42_c238_d2_s0	0.01	0	20	20	20	19.5
v56_c372_d2_s0	0.02	0.01	27	37	25	33
v72_c549_d2_s0	0.04	0.03	39	87	32	77.5
v90_c775_d2_s0	0.08	0.05	52	135	43	93.5
v110_c1056_d2_s0	0.24	0.16	158	282.5	99	179.5
v156_c1807_d2_s0	1.26	0.33	565	408	348	250
v210_c2850_d2_s0	0.79	1.09	135	1083	121	450
v272_c4233_d2_s0	1.67	4.1	179	3004	147	972.5

Table 31: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on GT_N problems.

File Name	Search Time		# of Decisions		# of Backtracks	
	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3	FDBK-V	FDBK-H3
v100_c100_d3_cs3_s1	0	0	53	34	0	0
v100_c1000_d3_cs3_s1	0	0	8	5	0	0
v200_c200_d3_cs3_s1	0.01	0.03	148	75	0	0
v200_c2000_d3_cs3_s1	0.01	0.02	17	11	0	0
v300_c300_d3_cs3_s1	0.02	0.06	248	110.5	0	0
v300_c3000_d3_cs3_s1	0.02	0.04	24	15	0	0
v500_c500_d3_cs3_s1	0.05	0.19	382	192	0	0

Table 32: Comparison of median of search time, number of decisions, number of backtracks done by FD-BK (VSIDS) and FD-BK (Heuristic 3) over 10 runs on random satisfiable problems.

Table 29, Table 30, Table 31 show that BK-V outperforms BK-H3 (i.e. (#s+#u)) on median search time, number of decisions and number of backtracks over 10 runs indicating that VSIDS is a better than Heuristic 3. Table 32 shows that BK-V outperforms BK-H3 i.e. (#s+#u) on median search time, over 10 runs. Table 32 shows that on random satisfiable problems BK-V produces more number of median decisions over 10 runs compared to BK-H3. Even when BK-V produces more median number of decisions over 10 runs compared to BK-H3, BK-V outperforms BK-H3 on mean search time due to low computational cost of VSIDS compared to H3.

Chapter 7

Conclusion

In this thesis we further extended the FD DPLL algorithm introduced by Sinha [23] and extended by Nagle [20] and Lal [15]. The thesis was aimed at answering the question whether the watched literal approach introduced by the landmark Boolean SAT solver Chaff [18] wins over the bookkeeping approach when applied in the FD SAT setting. We also wanted to investigate why this should be so. To do this end, we put a great deal of effort into the development of the high-level descriptions that appear here as Algorithms 4 and 5. Also, this thesis was aimed at comparing the performance of the VSIDS heuristic introduced by Chaff [18] with different bookkeeping heuristics.

We compared the performance of the watched literals approach with bookkeeping approach with both having the same heuristic and found that watched literals wins over bookkeeping approach on problems that require great deal of backtracking, whereas bookkeeping beats watched literals approach on easier problems.

We also compared different bookkeeping heuristics relative to VSIDS and found that VSIDS performs very well on all sorts of problems. This is probably due to the fact that it is dynamic yet cheap and also it makes good use of the learned clauses.

7.1 Future Work

1. *Further work on Heuristics*: Heuristics still seem important while searching for a model. More work is needed to compare the tradeoffs between fast implementation and stronger statistics for heuristics. It would be good to try heuristics based on the current stats augmented with some weighing function for learned clauses as in VSIDS.

2. *Random Restarts*: Random restarts have been incorporated in recent Boolean SAT solvers. They periodically empty the current partial interpretation and start all over again, ensuring that learned clauses are utilized from the beginning and may take the search into new space. This is particularly important to investigate the widely observed phenomenon of tail-heavy distributions of solution times on hard SAT problems.

3. Integration with CCalc: CCalc is software that implements reasoning about actions and planning. Currently it uses Boolean SAT solvers to solve the problem, despite allowing input to be finite domain. Modifying CCalc so that instead of using a Boolean SAT solver it uses a FD SAT solver could be an extension that is useful.

4. *More Finite Domain Structured Problems*: We need more structured problems both unsatisfiable as well as satisfiable to test our solvers.

5. *Harder random FD SAT problems, especially unsatisfiable ones.*

6. *Further work on Clause Learning and Non-Chronological Backtracking*: Different clause learning techniques might help in improving the performance of the solver. It would be also interesting to consider other choices for backtracking level.

Bibliography

[1] Ansotegui C., and Manya F., "Mapping problems with finite-domain variables into problems with Boolean variables". In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, 2004.

[2] Bayardo R., and Schrag R., "Using CSP look-back techniques to solve real-world SAT instances". In *National Conference on Artificial Intelligence (AAAI)*, 1997.

[3] Beame P., Impagliazzo R., Pitassi T., and Segerlind N., "Memoization and DPLL: Formula caching proof systems." In *Proceedings 18th Annual IEEE Conference on Computational Complexity*, pages 225–236, 2003.

[4] Beame P., Kautz H., and Sabharwal A., "Understanding the Power of Clause Learning". In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, 2003.

[5] Beame P., Kautz H., and Sabharwal A., "Towards Understanding and Harnessing the Potential of Clause Learning". In *Journal of Artificial Intelligence Research*, pages 319 – 351, 2004.

[6] Ben-Sasson E., Impagliazzo R., and Wigderson A., "Near-optimal separation of tree-like and general resolution." In *Electronic Colloquium in Computation Complexity*, 2000.

[7] Bonet M., and Galesi N., "Optimality of size-width tradeoffs for resolution". In *Computational Complexity*, 10 (4), pages 261–276, 2001.

[8] Brayton R., and Khatri S., "Multi-valued logic synthesis". In *Proceedings of the International Conference on VLSI Design*, 1999.

[9] Davis M., and Putnam H., "A Computing Procedure for Quantification Theory". In *Journal of the Association for Computing Machinery*, 1960.

[10] Davis M., Logemann G., and Loveland D., "A machine program for theorem-proving". In *Communications of the ACM*, pages 394–397, July 1962.

[11] Giunchiglia E., Lee J., McCain N., Lifschitz V., and Turner H., "Nonmonotonic causal theories". In *Artificial Intelligence 2004*, 153 (1-2), pages 49–104, March 2004.

[12] Goldberg E., and Novikov Y., "BerkMin: A Fast and Robust SAT Solver". In *Design Automation and Test in Europe (DATE)*, 2002.

[13] Gomes C., Kautz H., Sabharwal A., and Selman B. "Handbook of Knowledge Representation", In *Foundations of Artificial Intelligence*, 3, Editors: van Harmelen F., Lifschitz V., and Porter B., Elsevier, pages 89-134, 2008.

[14] Krishnamurthy B., "Short proofs for tricky formulas". In *Acta Informatica*, pages 253–274, 1985.

[15] Lal H., "A Finite Domain Satisfiability Solver with Clause Learning and Non-Chronological Backtracking". Master's thesis, University of Minnesota Duluth, July 2005.

[16] Liu C., Kuehlmann A., and Moskewicz M., "CAMA: A Multi-Valued Satisfiability Solver". In *International Conference on Computer Aided Design*, pages 326 – 333, November 2003.

[17] Mitchell D., "A SAT Solver Primer". In *Proceedings of Bulletin of the EATCS*, pages 112 – 132, February 2005.

[18] Moskewicz M., Madigan C., Zhao Y., Zhang L., and Malik S., "Chaff: Engineering an Efficient SAT Solver". In *Proceedings of 38th Design Automation Conference (DAC2001)*, June 2001.

[19] Nadel A., "Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations". Master's thesis, Hebrew University, November 2002.

[20] Nagle A., "A Finite Domain Solver". Master's thesis, University of Minnesota Duluth, August 2004.

[21] Ryan L., "Efficient Algorithms for Clause-Learning SAT Solvers". Master's thesis, Simon Fraser University, February 2004.

[22] Silva J., and Sakallah K., "GRASP – A New Search Algorithm for Satisfiability". In *Proceedings of the International Conference on Computer-Aided Design*, November 1996.

[23] Sinha S., "A Finite Domain Solver". Master's thesis, University of Minnesota Duluth, August 2003.

[24] Zhang L., Madigan C., and Moskewicz M., "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver". In *International Conference on Computer-Aided Design*, November 2001.