A Finite Domain Satisfiability Solver

with Clause Learning

and Non-Chronological Backtracking

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Hemal A. Lal

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

July 2005

UNIVERSITY OF MINNESOTA

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

HEMAL A. LAL

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 Adviser

———————————————————

Signature of Faculty Advisor

————————————

Date

GRADUATE SCHOOL

Abstract

Boolean Satisfiability (SAT) is a well known NP-complete problem that has recently received much attention due to increasing practical application in various fields such as Artificial Intelligence, Group Theory, Formula Verification, Electronic Design Automation, Logic Synthesis, etc. A search algorithm to solve this problem was proposed by Davis, Putnam, Logemann, and Loveland (DPLL) some four decades ago. Although DPLL is an old algorithm, most of the current state-of-the-art SAT solvers are based on it. Recent additions to DPLL such as clause learning, non-chronological backtracking, and random restart have improved the capability of problem solving.

Finite Domain Satisfiability is a generalization of Boolean Satisfiability in which the size of the domain of a variable may be greater than two. Many practical applications fall under this category, but to solve them, traditionally they are encoded into Boolean format and solved using a Boolean solver. This not only obscures the structure of the problem but also increases the size of the problem, hence making it harder to solve.

In this thesis, we develop a SAT solver that can directly be used to solve a Finite Domain SAT problem. This work is a continuation of work done by Sinha, and Nagle. In this thesis, we adapt and implement the state-of-the-art techniques clause learning and non-chronological backtracking.

The culmination of this thesis is a Finite Domain solver whose performance is compared with the state-of-the-art Boolean SAT solver `zchaff` as well as a state-of-the-art Finite Domain solver called `CAMA`, developed independently by Liu et al.

Contents

List of Figures

List of Tables

1 Introduction

Boolean satisfiability (SAT) is a classic NP-complete problem. A Boolean SAT problem instance is a Boolean expression made up of AND, OR, NOT, variables, and parentheses; given a Boolean expression, the problem is to determine whether there is an interpretation (an assignment of TRUE or FALSE to variables) that makes the expression true. Despite the fact that its known algorithmic solutions have an exponential worst case complexity, Boolean SAT has found a wide range of applications in various fields, such as Artificial Intelligence, Group Theory, Formula Verification, Electronic Design Automation (EDA), Logic Synthesis, etc. A variety of interesting classes of problems in these fields can be reduced to Boolean SAT and solved relatively effectively using SAT solvers [10, 13, 15, 16, 19, 24]. Many algorithms have been proposed for Boolean SAT solvers; most are variations of the Davis, Putman, Logemann, and Loveland (DPLL) [7] search-based algorithm [10, 16, 19, 22]. Notable recent developments are clause learning, non-chronological backtracking, and random restart, which have improved observed solving times tremendously [2, 5, 11, 14, 18, 23]. Publicly available Boolean SAT solvers include `zchaff` [23], GRASP [19], and SATO [22].

Many of the above mentioned applications are more naturally encoded as Finite Domain SAT problems, a natural extension of Boolean SAT in which variables may have finite domains other than $\{\mathbf{t}, \mathbf{f}\}$. One of the reasons for this is that Finite Domain SAT better retains the structure of the problem while also requiring fewer variables and clauses. To solve these kinds of problems, these days they are encoded into Boolean SAT and solved using Boolean SAT solvers instead of being solved directly using a Finite Domain SAT solver. In this thesis we build a Finite Domain solver which extends work done by Sinha [20] and Nagle [17] on Finite Domain solvers by incorporating clause learning and non-chronological backtracking.

1.1 Finite Domain Satisfiability

Finite Domain SAT is a natural extension of Boolean SAT whereby variables in a theory can be mapped to values other than $\mathbf{t}$ or $\mathbf{f}$. A *signature* in Finite Domain SAT is a finite set of symbols, called *variables*, where each variable V is associated with a finite set of symbols *dom(V)*, called its *domain*. An *atom* is an expression of the form $V = x$ where V is a variable and $x \in dom(V)$. A *literal* is an atom $V = x$ or its *negation*, written $V \neq x$. A *clause* is a finite set of literals. A *theory* is a finite set of clauses.

As an example, consider a signature with 3 variables, $\{A, B, C\}$, each variable having a domain of $\{0, 1, 2\}$. Then the following set of 5 clauses is an example of a theory in this signature.

$$\{\{A=0, B\neq1\}, \{A=1, B\neq1\}, \{A=2, B=1\}, \{A\neq2, B=2\}, \{A=0, B\neq2, C=2\}\} \qquad (1)$$

The Finite Domain SAT problem is the problem of deciding whether there is an assignment of values to variables that "satisfies" a theory such as (1). We now make the notion of "satisfaction" precise.

An *interpretation* I is a function mapping each variable of the signature to a value from its domain. An interpretation I *satisfies* an atom $V=x$ if $I(V) = x$, and *satisfies* its negation $V\neq x$ 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. The Finite Domain SAT problem then is to determine whether a given theory has a model. In applications, it is typically useful to also return a model if one exists.

For example, the interpretation that satisfies the atoms $A=2, B=2, C=2$ is a model of (1). In fact, this is the only model of the theory.

Consider another example using the same signature as above.

$$\{\{A=0\}, \{A=1, B=2\}, \{A=2, B=1\}\} \qquad (2)$$

This theory does not have any interpretation that satisfies it. We say that (2) is unsatisfiable.

Notice that there is no interpretation that can satisfy the empty clause (i.e. $\{\ \}$), while on the other hand, any clause that contains an atom and its negation is satisfied by every interpretation

Boolean SAT is the special case of Finite Domain SAT in which the domain of each variable has size two, usually expressed as $\{\mathbf{t}, \mathbf{f}\}$. If we agree to represent $V=\mathbf{t}$ as V and $V=\mathbf{f}$ as $\neg V$, we have a standard Boolean SAT representation.

Many real world problems are naturally encoded as Finite Domain SAT problems, because real world problems often involve variables whose values can be understood to range over a finite (unordered) domain.

1.2 Motivation and Related Work

Many SAT problems such as automated planning, graph coloring, etc., are more naturally expressed as Finite Domain SAT problems, but due to lack of research and unavailability of Finite Domain solvers, these problems are instead encoded into Boolean SAT problems and are solved using state-of-the-art Boolean solvers. There are a number of available translation schemes such as Full Logarithmic Encoding [4], Full Regular Mapping [4], Linear Encoding [9], Quadratic Encoding [9], etc. Nonetheless, such encodings increase the size of the problem and also obscure the structure of the problem, thus making it harder to solve. This has led to our interest in Finite Domain solvers, which can be directly used to solve Finite Domain SAT problems.

Sinha's [20] work on Finite Domain solvers provides evidence that solving Finite Domain SAT problems using a Finite Domain solver may be advantageous. Her relatively naive implementation of a Finite Domain solver is competitive with and many times outperforms state-of-the-art Boolean solvers. (The comparisons are done by generating random satisfiable Finite Domain theories T and their Boolean translations $B(T)$; then comparing performance of Sinha's solver on T with performance of the Boolean solver on $B(T)$.)

Another related work done on Finite Domain solvers is by Nagle [17]. His work improved the expressiveness of Sinha's input language by including negation in the problems. He designed and implemented a solver that accepts and solves these kinds of problems, and provided further evidence of the potential usefulness of the approach.

Liu et al. [13] recently introduced CAMA, a Finite Domain solver that adapts recently developed speed-up techniques used in state-of-the-art Boolean solvers such as clause learning and non-chronological backtracking. Conflict-based clause learning is a technique whereby you learn new clauses from conflicts generated during the search for a model and direct your search further based on what you learned. The input language of CAMA is richer than ours in the sense that CAMA's variables take a set of values. An example of an atom in CAMA's format would be $V^{\{0,2,3\}}$, stating that variable V can take value 0 or 2 or 3. Note that in our language, this can be expressed, but requires replacing $V^{\{0,2,3\}}$ in a clause with a family of literals (either $\{V=0, V=2, V=3\}$ or $\{V \neq x \mid x \in dom(V), x \notin \{0,2,3\}\}$, whichever is smaller).

Frisch et al. [8] also worked on Finite Domain SAT problems. They present NB-Walksat, a Finite Domain solver that uses a stochastic local search method. This is an incomplete method; it does not have a systematic

process for searching. Incomplete methods cannot determine unsatisfiability but have proven to be quite good sometimes in solving hard satisfiable Boolean SAT problems. Also Frisch with Stock [21] worked on a Finite Domain solver NB-satz (which is complete). It is an extension of the publicly-available Boolean solver satz [6], for Finite Domain SAT problems. They generated random problems and used NB-satz on these problems and compared the results with satz run on the Boolean encoding of the problems. From the results, they conclude that NB-satz on many instances outperformed satz. Although the results reported are quite good, this solver does not have clause learning nor non-chronological backtracking, thus performing poorly on some classes of hard unsatisfiable problems. It uses the "Maximum occurrences in clauses of minimum size" (MOMS) heuristic from satz for variable selection. The author though reports that this heuristic when extended for Finite Domain is particularly brutal since it might select a wrong space to search.

Another extension of satz, MV-satz was developed independently by Ansotegui et al. [3]. This solver utilizes the optimized data structures and heuristics from satz. It adds extra filters that improve the heuristics for Finite Domain SAT problems. But, as with NB-satz, it does not have any clause learning nor non-chronological backtracking.

Another method for solving Finite Domain problems besides encoding it into Boolean format or using a Finite Domain SAT solver is to describe the problem as a constraint satisfaction problem (CSP) and use a CSP solver to solve it. Like Finite Domain problems, CSP problems have variables, domains for each variable, and a set of constraints. A solution to a CSP problem is an assignment of values to each variable for which all constraints are satisfied. Sinha [20] did some experiments with CSP solvers to analyze if they provided any advantage over Finite Domain SAT solvers. She reports that while CSP languages are typically much more expressive than hers (or CAMA's or ours), CSP solvers are not optimized to solve problems that are nicely expressed as Finite Domain SAT problems.

This thesis develops a Finite Domain solver which incorporates clause learning and non-chronological backtracking. It also provides evidence that the above mentioned techniques improve the efficiency of the solver. Although this solver is similar to CAMA in aspects of clause learning and non-chronological backtracking, it is rather different in another crucial respect. Specifically, CAMA uses the "two-watched-literal" approach introduced in state-of-the-art Boolean solver zchaff. In this approach the solver sacrifices some of its ability to compute accurate (dynamic) heuristics (to guide the search) in exchange for much greater speed in

execution of backtracking, while our solver utilizes a more exhaustive bookkeeping approach where we keep counts of occurrences of each literal in the theory and update the counts as the search proceeds. This gives us the ability to compute much more accurate dynamic heuristics. (Alam [1] has done some comparison between the two approaches and reports his findings.) From our experiments and findings we observe that bookkeeping approach often performs better than the "two-watched-literals", at least for the Finite Domain problems we consider. The Evaluation chapter presents some results of these experiments.

1.3   Contribution of this Thesis

In this thesis we extend work done by Nagle [17] and Sinha [20] on development of a Finite Domain SAT solver. Sinha's solver is based on a Finite Domain extension of the Boolean DPLL algorithm; Nagle's solver extends Sinha's solver by including better heuristics and also permitting *negative* literals of the form $V \neq x$ to occur in a theory. Although both of these solvers perform competitively well against the Boolean encoding of random satisfiable Finite Domain SAT problems run on state-of-the-art Boolean solvers, neither of them learns from conflicts that occur during the search. Zhang et al. [23] report that, for Boolean solvers at least, DPLL without clause learning works well with randomly generated problems but is less effective with structured problems.

In our experiments, we use not only randomly generated satisfiable problems (as Sinha and Nagle did) but also unsatisfiable problems and a variety of structured problems.

In our solver we incorporate clause learning and non-chronological backtracking (details in chapter 2). Comparison of experimental results with Sinha's and Nagle's Finite Domain solvers show that our solver outperforms both of these solvers on almost 90% of our test data sets. We also compare our solver's results with `CAMA` solver by comparing the run time and number of backtracks for each test problem. We observe that on randomly generated test problems we outperform `CAMA` on about 95% of problems, and on about 75% of structured problems. We use a state-of-the-art Boolean solver `zchaff` to compare our solver with because it is overall the best publicly available Boolean SAT solver. Comparison of experimental results between `zchaff` and our solver show that our solver is competitive with `zchaff` and often outperforms it.

1.4   Outline of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we present in detail our generalization of the DPLL algorithm for the purpose of Finite Domain SAT. In particular, it extends prior work by Sinha and Nagle by showing how to incorporate clause learning and non-chronological backtracking. Chapter 3 discusses implementation details. In Chapter 4, we present the evaluation methods used and report the experimental results. Finally, Chapter 5 concludes the thesis and suggests future work.

2 Finite Domain SAT

In this section we describe our Finite Domain DPLL algorithm. We begin with background knowledge concerning partial interpretations, unit propagation, and entailment. Then we discuss the algorithm itself, including the technique for analyzing conflicts and backtracking. Finally we discuss the heuristics used to help guide the search.

2.1 Partial Interpretation

A *partial interpretation* is a set P of literals, such that for every variable V,

1. P contains at most one element of $\{V = x \mid x \in dom(V)\}$, and

2. P does not contain all elements of $\{V \neq x \mid x \in dom(V)\}$, and

3. P does not contain an atom and its negation (i.e. both $V = a$ and $V \neq a$).

We say a partial interpretation P *satisfies* an atom $V = x$ if $V = x \in P$ and satisfies a literal $V \neq x$ if either $V \neq x \in P$ or $V = x' \in P$ with $x' \neq x$. A partial interpretation P *falsifies* a literal if it satisfies its complement. (The *complement* of $V = x$ is $V \neq x$, and the *complement* of $V \neq x$ is $V = x$). A partial interpretation *satisfies* a clause if it satisfies at least one literal from the clause, and it *satisfies* a theory if it satisfies all the clauses in the theory. A partial interpretation *falsifies* a clause if it falsifies all the literals in it, and it *falsifies* a theory if it falsifies at least one clause in the theory.

The conditions $(1) - (3)$ above on partial interpretation P guarantee its "consistency": the first condition maintains that at most one value is assigned to a variable, while condition (2) guarantees that there is at least one value that can be assigned to a variable and condition (3) guarantees that P does not simultaneously require and rule out an assignment. Thus, for any partial interpretation P there exists at least one interpretation I that "satisfies" P, that is, I satisfies every literal in P. It follows that if there is a partial interpretation P that satisfies a theory T, then there is also an interpretation I that satisfies T. In fact, every interpretation that satisfies P is also a model of T. On the other hand, if interpretation I satisfies T, then some partial interpretation P (that is satisfied by I) satisfies T. Therefore a theory is satisfiable if and only if some partial interpretation satisfies it.

Our DPLL algorithm is essentially a depth first search in the space of partial interpretations, looking for one that satisfies the theory or until there are no more partial interpretations left to be searched. If there exists a partial interpretation that satisfies the theory, the theory is satisfiable. Moreover, it is straightforward to "extract" a model of the theory from the partial interpretation.

2.2   Unit Propagation

A key element of the DPLL algorithm − one that makes it much more interesting and effective than simple depth-first search in the space of partial interpretations − is the incorporation of a fast inference method called "unit propagation".

A *unit clause* is a clause that is a singleton. Notice that if T contains a unit clause $\{V\!=\!a\}$, then any model of T must map V to a. Consequently, we can safely assume that we are looking for a partial interpretation that is a superset of $\{V\!=\!a\}$. Next notice that if T also contains a clause $\{V\!\neq\!a, W\!=\!b\}$ then any partial interpretation that satisfies T must be in fact a superset of $\{V\!=\!a, W\!=\!b\}$. These observations motivate the following.

Let P be a partial interpretation. A clause is *unit* w.r.t. P if it is not satisfied by P and has exactly one unassigned (neither satisfied nor falsified by P) literal; we call this literal the *unit literal*. To satisfy a unit clause we need to satisfy the unit literal. To find a superset of P that satisfies a theory we need to satisfy all the unit clauses in the theory w.r.t. P. When we satisfy a unit literal we update P by adding the literal to the partial interpretation. Thus, the process of satisfying a unit clause may lead to additional unit clauses (w.r.t. P) in the theory; hence the name *Unit Propagation*.

Unit propagation may also generate falsified clauses, causing a "conflict" to occur. At that point we need to backtrack in our search (details follow later).

Let's look at an example of unit propagation. Let the signature be $\{A, B, C\}$ and the domain of each variable be $\{0, 1, 2\}$. The initial partial interpretation P is $\{\}$, and the theory consists of the following clauses.

$$\text{Clause } 1 : \{A\!=\!0\}$$
$$\text{Clause } 2 : \{A\!=\!2, B\!=\!2\}$$
$$\text{Clause } 3 : \{A\!\neq\!1, C\!=\!2\}$$

$$\text{Clause 4}: \{B=1, C \neq 1\}$$
$$\text{Clause 5}: \{A=1, B=1, C \neq 2\}$$

Clause 1 is a unit clause w.r.t. P, so we satisfy $A = 0$ by adding it to P. So P becomes $\{A = 0\}$ and, intuitively, the theory then looks like

$$\text{Clause 1}: \text{``satisfied''}$$
$$\text{Clause 2}: \{B=2\} \text{ (falsified } A=2\text{)}$$
$$\text{Clause 3}: \text{``satisfied'' (due to } A \neq 1\text{)}$$
$$\text{Clause 4}: \{B=1, C \neq 1\}$$
$$\text{Clause 5}: \{B=1, C \neq 2\} \text{ (falsified } A=1\text{)}$$

Now Clause 2 is a unit clause w.r.t. P, so we satisfy it. Then $P = \{A=0, B=2\}$, and the theory becomes (intuitively)

$$\text{Clause 1}: \text{``satisfied''}$$
$$\text{Clause 2}: \text{``satisfied''}$$
$$\text{Clause 3}: \text{``satisfied''}$$
$$\text{Clause 4}: \{C \neq 1\} \text{ (falsified } B=1\text{)}$$
$$\text{Clause 5}: \{C \neq 2\} \text{ (falsified } B=1\text{)}$$

Now clauses 4 and 5 are unit clauses w.r.t. P. They are handled one after another since satisfying either one does not affect the other. After satisfying clauses 4 and 5, $P = \{A=0, B=2, C \neq 1, C \neq 2\}$. Note that by satisfying clauses 4 and 5 we have ruled out the values 1 and 2 for variable C, and since *dom(C)* $= \{0, 1, 2\}$, we know that we must assign 0 to C. This is called "*Entailment*" (discussed next) and the atom $C=0$ is called an "*Entailed atom*".

2.3 Entailment

In addition to unit propagation, we can also sometimes infer a variable assignment from a partial interpretation P itself.

An atom *Entailment* occurs when for some variable V there is a value $x \in dom(V)$ such that the atom $V = x$ $\notin P$ and for all $x' \in dom(V)$ where $x' \neq x$, $V \neq x' \in P$. The atom $V = x$ is said to be *entailed* by P and is called an "*Entailed atom*". The entailed atom is satisfied and added to P. Notice that if P is a partial interpretation (and so satisfies the relevant conditions) and $V = x$ is entailed by P, then $P \cup \{V = x\}$ is also a partial interpretation. Moreover, any interpretation that "satisfies" P must also satisfy $P \cup \{V = x\}$. In this sense, adding an entailed atom to P is both safe and, roughly speaking, unavoidable.

2.4   Finite Domain SAT - Algorithm

The above three sections provided us with the background knowledge we need. Let us now look at the Extended Finite Domain DPLL algorithm.

We begin with a rough description of the algorithm. Begin with a theory T and an empty partial interpretation P. Perform any unit propagation and atom entailments. At this point P consists of literals that must be true in any partial interpretation that satisfies T. If at this point P satisfies T, then we are done. On the other hand, if at this point P falsifies T, we conclude that T is unsatisfiable. Otherwise, some search is necessary. So choose some literal to add to P. (Chosen so that the result is a partial interpretation.) We will refer to a literal so chosen as a "decision" literal. As a consequence of adding this decision literal to P, we may have new unit clauses (or entailed atom). Perform unit propagation and atom entailment. Again, if at this point if P satisfies T, we are done. (Return P.) If P neither satisfies nor falsifies T, we need to choose another decision literal. If P falsifies T, then we have a "conflict" and need to backtrack in order to search another part of the space of partial interpretations. And so on.

This description of the algorithm gives the basic idea, but there are quite a few details to fill in.

In our precise description of algorithm, we will imagine that a literal in P has 3 associated "fields": `level`, `index`, and `reason`. The `level` field tells us at what depth in the search space of partial interpretations was the literal added to the partial interpretation P. That is, `L.level` is simply the number of decision literals in P just after L was added to P. The `index` field tells us the order in which literals were added. That is, `L.index` is simply the number of literals in P just after L was added to P. The `reason` field records the "reason" literal L was added to P. If L was obtained by unit propagation, then `L.reason` is the clause in T in which L was found to be the unit literal. If L is an atom $V = a$ that was obtained by atom

entailment, then L.reason is the clause $\{V\!=\!x \mid x \in dom\{V\}\}$. Otherwise, L was obtained by choice and L.reason $=$ "D" to indicate that L is a decision literal.

Below are descriptions of several auxiliary functions used by the main algorithm.

level(P): This function returns the number of decision literals in P.

addLiteral(L, reason): This function adds the literal L to partial interpretation P and it sets the literal's index, level, and reason fields appropriately. In particular, it sets L.reason to reason.

choose_next_branch_literal(): This function returns a literal L such that (i)$L \notin P$ and (ii)$P \cup \{L\}$ is a partial interpretation. This function is used to obtain a decision literal when one is needed. Typically some care is taken to try to choose well. To this end, heuristics are used to select from among the available literals. Such heuristics are discussed further in Section 2.6.

AnalyzeConflict(C): This function is called when it is discovered that the current partial interpretation falsifies some clause C in T. It is at this point that clause learning occurs. The function returns the learned clause. (The clause learning algorithm will be described in detail a little later.)

getBacktrackLevel(C): This function returns the minimal level (at which the literal was falsified) from among the literals in a (learned) clause C which is not the current level. It uses the whyFalse(L).level function to get the level at which L was falsified. (After we backtrack to that level, the clause C will be unit w.r.t. the (new) current partial interpretation. This guarantees that we make immediate use of the learned clause via unit propagation when we continue after backtracking.)

undo(level): This function restores P to its state just before the decision at level level+1 was made. That is, it removes from P all literals whose level is greater than level.

Figure 1 shows pseudocode of the algorithm.

The functions AnalyzeConflict(C) and getBacktrackLevel(C) in turn make use of another auxiliary function whyFalse(L), described below.

whyFalse(L): This function returns the literal L' with earlier index among those literals in P that are inconsistent with L. For example, using the format $V = x @ N$ (where $V = x$ is a literal with index N), let $P = \{C\!=\!1@1, A\!\neq\!0@2, B\!=\!2@3, A\!=\!1@4\}$. If L is $A\!=\!0$, then function whyFalse(L) would return $A\!\neq\!0@2$ instead of $A\!=\!1@4$.

```
Global information
P is a global variable whose value is the current partial interpretation.
  DPLL(T){
    P ← {}
    while(true){
      if P satisfies T
        return SATISFIABLE
      else if there is a clause C in T s.t. P falsifies C
        if level() = 0
         return UNSATISFIABLE
        else
         learnedClause ← AnalyzeConflict(C)
         level ← getBacktrackLevel(learnedClause)
         undo(level)
         T ← T ∪ {learnedClause}
      else if there is an entailed atom V=a w.r.t. P
        addLiteral(V=a, {V=x : x ∈dom(V)})
      else if there exists a unit clause C with unit literal L
          in T with respect to P
        addLiteral(L,C)
      else
        branchLiteral ← choose_next_branch_literal()
        addLiteral(branchLiteral, ``D'')
    }
  }
```

P is a global variable whose value is the current partial interpretation.

Figure 1: Finite Domain SAT - Algorithm pseudocode

```
level(){
  return size({L ∈ P | L.reason = ''D''})
}
addLiteral(L, reason){
  P ← P ∪ {L}
  L.index ← size(P)
  L.level ← level(P)
  L.reason ← reason
}
undo(level){
  P ← P − {L ∈ P | L.level > level}
}
getBacktrackLevel(C){
  levels ← {whyFalse(L.level) | L ∈ C }
  levels ← levels − {max(levels)}
  If levels = ∅
    return 0
  return max(levels)
}
```

Figure 2: Finite Domain SAT - Auxiliary function pseudocode

Figure 2 shows pseudocode for all auxiliary functions called directly from the main algorithm, except function `AnalyzeConflict(C)`, which is described in section 2.5.

2.5 Conflict Analysis and Non-Chronological Backtracking

A conflict occurs when a clause is falsified by the current partial interpretation. (That is, all the literals in the clause are falsified). Conflict analysis takes the falsified clause and considers the assignments that led to the conflict, and learns a clause using the *reasons* for these assignments. The learned clause is one that is entailed by the theory. That is, any interpretation that satisfies the theory also satisfies the learned clause. For example, consider the theory and the learned clause shown below. (Signature has 3 variables, $\{A, B, C\}$, with domain of each variable $\{0, 1, 2\}$.)

c1 : $\{A=0, B=1, C\neq2\}$
c2 : $\{A=0, B=1, C=2\}$
learned clause : $\{A=0, B=1\}$

Observe that the learned clause is entailed by c1 and c2. The reason we want the learned clause to be entailed by the theory is that when we add the learned clause to theory we do not want to change the models of the theory.

The algorithm is designed so that, once the learned clause is added and we backtrack to the appropriate level, this clause will be unit (w.r.t. the current partial interpretation). In this way we make immediate use of the learned clause. Of course the learned clause may also affect later portions of the search, after subsequent backtracking. (In fact, termination of the algorithm depends crucially on this.)

To learn a clause from conflict, conflict analysis uses resolution. The idea of resolution is to take two clauses and merge them forming a new clause while removing a literal and its complement. Take two clauses of the form $\{V = x\} \cup C$ (*) and $\{V \neq x\} \cup C'$ (**). Resolution performed on (*) and (**) results in $C \cup C'$. Now consider an interpretation I that satisfies both (*) and (**). Of course I must falsify one of $V = x$, $V \neq x$. If I falsifies $V = x$, it must satisfy C, since it satisfies (*). On the other hand, if I falsifies $V \neq x$, it must satisfy C', since it satisfies (**). Either way, I satisfies $C \cup C'$. The above discussion shows that the original clauses (*) and (**) entail the resolvent (the clause obtained by resolving (*) and (**)). Therefore adding the resolvent to a theory containing (*) and (**) does not affect its models. Thus resolution is a

14

sound inference method, and we can safely use it to help guide our search for a model.

The version of resolution we actually use in our algorithm, as defined in function `resolve(C, L, C′)`, is slightly more general and slightly stronger. Notice that the prior discussion of soundness of resolution involved considering cases based on whether a certain literal is satisfied by a given interpretation. In our function, the second 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 clause $C′$ that are satisfied by at least one interpretation that *also satisfies* L. Consider the following example. (Signature has 3 variables, $\{A, B, C\}$, with domain of each variable $\{0, 1, 2\}$.)

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

$L : C=2$

$C′ : \{A=1, B=0, C\neq 2\}$

$R : \{A=0, A=1, B=0, B=2\}$

Consider another example using the same signature.

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

$L : C=2$

$C′ : \{A=1, B=0, C=0, C=1\}$

$R : \{A=0, A=1, B=0, B=2\}$

This example shows how resolution can remove more than one literal from a clause.

To show that `resolve(C, L, C′)` is sound, we need to show that $\{C, C′\}$ entails (the clause returned by) `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 satisfies L. Since I satisfies $C′$, I satisfies one of the literals in $C′$ that can be satisfied by some interpretation that satisfies L. Consequently, I satisfies `resolve(C, L, C′)`.

Case 2: Otherwise. Since I satisfies 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′)`.

```
AnalyzeConflict(C){
    If Potent(C)
        return C
    L ← maxLit(C)
    L' ← whyFalse(L)
    return AnalyzeConflict(resolve(C, L, L'.reason))
}
```

Figure 3: Finite Domain SAT - Conflict Analysis

So we do conflict analysis when some clause $C \in T$ is falsified by the current partial interpretation P. Given the clause C falsified w.r.t. P, we use resolution to derive the learned clause. We take clause C and get the latest literal L falsified in C. This literal L is falsified w.r.t. P because there is a literal $L' \in P$ which is inconsistent with L. The "reason" for L' (that is, L'.reason) is a clause. (We know L'.reason is a clause − and not "D" − because clause C must have been unit w.r.t. P just before L' was added to P in order for C to have been falsified by the addition of L' to P. And of course clause C cannot have been unit at the time a decision was made; we would have done unit propagation instead.) We resolve clause C and L'.reason with respect to literal L. We check the result to see if it has exactly one literal falsified at the current level. If so, we are done. If not, we will repeat this process until we have obtained a resolvent with exactly one literal that was falsified at the current level. This will then be the learned clause, and we will backtrack (removing literals from P) to a point where the resolvent is unit (and continue from there).

Below are some descriptions for functions used in the pseudocode for Conflict Analysis.

Potent(C) : This function returns true if in the clause C there is exactly one literal L for which whyFalse(L).level is the current level. Otherwise returns false.

maxLit(C) : This function returns a literal L from C for which whyFalse(L).index is maximal.

The pseudocode for Conflict Analysis is shown in Figure 3.

We decide the level to backtrack to using the clause returned by the call to AnalyzeConflict(C). The getBacktracklevel(C) function in Figure 2 gives a pseudocode of how it is done. We then use function undo() to backtrack (by removing some of the literals from P) to the earliest point at which (1)

16

the learned clause is unit (w.r.t. the new P), and (2) no unit propagation and atom entailments that are still "valid" are undone.

Looking at an example, if the learned clause is (2=0@2 1=3@4 3=0@6) (where the number after @ represents the level at which the literal was falsified) and current level is 6 the function returns 4 as the level to backtrack to. We remove from P all literals whose level is greater that 4. (Notice that we thus retain not only the decision literals at level 4, but also all level 4 unit literals and entailed atoms.) At this level the learned clause is unit, which leads the search into a new space of partial interpretations.

2.6   Heuristic

In this section we discuss the heuristic used in our implementation of the Finite Domain SAT algorithm to pick branching literals. (The function `choose_next_branch_literal()` in the pseudocode implements this idea.)

The heuristic selects an atom which has the maximal value of (*satisfaction - falsification*) where *satisfaction* is the number of clauses that would become satisfied if the atom is satisfied, and *falsification* is the number of not yet satisfied (nor falsified) clauses whose count of unassigned literals would be reduced due to the atom being satisfied.

Let's look at an example. The signature has 2 variables $\{A, B\}$ with domain $\{0, 1, 2\}$. The theory is

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

The heuristic will pick $A=0$ with heuristic value 2-1=1.

Although we don't report other heuristics in this thesis, we have tried numerous heuristics and have so far not found one that performs better overall. This heuristic is due to Nagle [17] who reports similar results to ours. Alam [1] reports in detail various comparisons for heuristics, including the one that we have used.

There are of course many possible heuristics that we have not explored. In particular it may be interesting to consider more expensive-to-compute heuristics, such as those used in Boolean solver satz, which was

adapted for use in Finite Domain solvers NB-satz and MV-satz, as mentioned previously.

2.7 Examples

We use the following examples to show how the algorithm works. Let the signature be $\{A, B, C, D\}$ and the domain of each variable be $\{0, 1, 2\}$.

Let P_{index} be the partial interpretation displayed with literals subscripted by their indexes.

Let P_{level} be the partial interpretation displayed with literals subscripted by their levels.

Let P_{reason} be the partial interpretation displayed with literals subscripted by their reasons.

2.7.1   Example 1 - Conflict at level 0 : Unsatisfiable instance

$c1 : \{A=0\}$

$c2 : \{A=1, B=1\}$

$c3 : \{A=2, B=2\}$

$P_{index} = \{\}$

$P_{level} = \{\}$

$P_{reason} = \{\}$

Perform unit propagation - clause c1 is a unit clause so we satisfy the unit literal $A=0$. Then we have the following:

$c1 : $ "satisfied"

$c2 : \{B=1\}$ ("falsified" $A=1$)

$c3 : \{B=2\}$ ("falsified" $A=2$)

$P_{index} = \{A=0_1\}$

$P_{level} = \{A=0_0\}$

$P_{reason} = \{A=0_{c1}\}$

Perform unit propagation - now we have two clauses that are unit, c2 and c3. Let's satisfy c2 first.

c1 : "satisfied"

c2 : "satisfied"

c3 : {} ("falsified" $A=2$, $B=2$)

$P_{index} = \{A=0_1, B=1_2\}$

$P_{level} = \{A=0_0, B=1_0\}$

$P_{reason} = \{A=0_{c1}, B=1_{c2}\}$

Now there is a "conflict" in theory, and since the current level is 0 we return the result UNSATISFIABLE.

2.7.2 Example 2 - Simple satisfiable instance

c1 : $\{A=0\}$

c2 : $\{A=1, B=1, C\neq0\}$

c3 : $\{A=2, B=1, C=0\}$

c4 : $\{B=2, C=0\}$

$P_{index} = \{\}$

$P_{level} = \{\}$

$P_{reason} = \{\}$

Perform unit propagation - clause c1 is a unit clause so we satisfy the unit literal $A=0$. Then we have the following:

c1 : "satisfied"

c2 : $\{B=1, C\neq0\}$ ("falsified" $A=1$)

c3 : $\{B=1, C=0\}$ ("falsified" $A=2$)

c4 : $\{B=2, C=0\}$

$P_{index} = \{A=0_1\}$

$P_{level} = \{A=0_0\}$

$P_{reason} = \{A=0_{c1}\}$

19

There are no unit clauses nor any atom entailment. Also there is no conflict thus far, so we need to choose some literal to add to P and proceed the search further. Our heuristic ranks $B=1$ and $C=0$ best. Let's say we picked $B=1$.

  c1 : "satisfied"

  c2 : "satisfied"

  c3 : "satisfied"

  c4 : $\{C=0\}$ ("falsified" $B=2$)

  $P_{index} = \{A=0_1, B=1_2\}$

  $P_{level} = \{A=0_0, B=1_1\}$

  $P_{reason} = \{A=0_{c1}, B=1_D\}$

Now we have a unit clause, c4. Perform unit propagation and satisfy literal $C=0$. The partial interpretation P now satisfies the theory, we return SATISFIABLE.

2.7.3 Example 3 - Clause learning and Non-Chronological Backtracking

Let the signature be $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ and the domain of each variable be $\{0, 1, 2, 3\}$. Assume that the algorithm has been partially executed, and the "relevant" part of T at this point is the following:

  c11 : $\{1=1, 3=0\}$ ("falsified" $2=1$)

  c12 : $\{1=2, 4=1\}$ ("falsified" $5 \neq 1$)

  c13 : $\{3=1, 6=2\}$ ("falsified" $5=0$)

  c14 : $\{4=3, 6=2\}$ ("falsified" $2 \neq 0$)

  c15 : $\{8=0, 6=0\}$ ("falsified" $7=0$)

w.r.t. partial interpretation:

  $P_{index} = \{2=0_0, 7=2_1, 9=0_2, 10=1_3, 11=2_4, 5=1_5, 12=0_6\}$

  $P_{level} = \{2=0_0, 7=2_1, 9=0_1, 10=1_2, 11=2_2, 5=1_3, 12=0_4\}$

  $P_{reason} = \{2=0_{c0}, 7=2_D, 9=0_{c1}, 10=1_D, 11=2_{c2}, 5=1_D, 12=0_D\}$

Let's pick $8\!=\!1$ as our decision literal. (Here we'll ignore the role of our heuristic in order to simplify our demonstration of learning and backtracking.) By satisfying the decision literal we get the unit clause c15, below.

c15 : $\{6\!=\!0\}$ ("falsified" $7\!=\!0$, $8\!=\!0$)

Satisfy the unit clause by satisfying the unit literal, $6\!=\!0$.
We perform unit propagation and get two unit clauses, they are c13 and c14 with unit literals $3\!=\!1$ and $4\!=\!3$ respectively.

c13 : $\{3\!=\!1\}$ ("falsified" $5\!=\!0$, $6\!=\!2$)
c14 : $\{4\!=\!3\}$ ("falsified" $2\!=\!0$, $6\!=\!2$)

Let us start by satisfying c13 and then followed by c14. At this point the theory looks like the following,

c11 : $\{1\!=\!1\}$ ("falsified" $2\!=\!1$, $3\!=\!0$)
c12 : $\{1\!=\!2\}$ ("falsified" $5\!\neq\!1$, $4\!=\!1$)
c13 : "satisfied"
c14 : "satisfied"
c15 : "satisfied"

w.r.t. partial interpretation:

$$P_{index} = \{2\!=\!0_0, 7\!=\!2_1, 9\!=\!0_2, 10\!=\!1_3, 11\!=\!2_4, 5\!=\!1_5, 12\!=\!0_6, 8\!=\!1_7, 6\!=\!0_8, 3\!=\!1_9, 4\!=\!3_{10}\}$$
$$P_{level} = \{2\!=\!0_0, 7\!=\!2_1, 9\!=\!0_1, 10\!=\!1_2, 11\!=\!2_2, 5\!=\!1_3, 12\!=\!0_4, 8\!=\!1_5, 6\!=\!0_5, 3\!=\!1_5, 4\!=\!3_5\}$$
$$P_{reason} = \{2\!=\!0_{c0}, 7\!=\!2_D, 9\!=\!0_{c1}, 10\!=\!1_D, 11\!=\!2_{c2}, 5\!=\!1_D, 12\!=\!0_D, 8\!=\!1_D, 6\!=\!0_{c15}, 3\!=\!1_{c13},$$
$$4\!=\!3_{c14}\}$$

We see that we have two new unit clauses, c11 and c12. Let us satisfy c12. By adding $1\!=\!2$ to the partial interpretation we have a conflict. c11 is falsified w.r.t. P. The conflicting clause is c11 and we need to analyze the conflict, learn and backtrack. At this point, P looks like the following,

$$P_{index} = \{2\!=\!0_0, 7\!=\!2_1, 9\!=\!0_2, 10\!=\!1_3, 11\!=\!2_4, 5\!=\!1_5, 12\!=\!0_6, 8\!=\!1_7, 6\!=\!0_8, 3\!=\!1_9, 4\!=\!3_{10}, 1\!=\!2_{11}\}$$
$$P_{level} = \{2\!=\!0_0, 7\!=\!2_1, 9\!=\!0_1, 10\!=\!1_2, 11\!=\!2_2, 5\!=\!1_3, 12\!=\!0_4, 8\!=\!1_5, 6\!=\!0_5, 3\!=\!1_5, 4\!=\!3_5, 1\!=\!2_5\}$$
$$P_{reason} = \{2\!=\!0_{c0}, 7\!=\!2_D, 9\!=\!0_{c1}, 10\!=\!1_D, 11\!=\!2_{c2}, 5\!=\!1_D, 12\!=\!0_D, 8\!=\!1_D, 6\!=\!0_{c15}, 3\!=\!1_{c13},$$
$$4\!=\!3_{c14}, 1\!=\!2_{c12}\}$$

We call `AnalyzeConflict(`C`)` function passing "c11" as an argument. (Note that what we pass in is the *original* c11, $\{1\!=\!1, 2\!=\!1, 3\!=\!0\}$.) The `Potent` function returns false when c11 is passed to it, since there are two literals in c11 that were falsified at the current level. So we get a literal L from c11 whose falsifying literal has maximal index. In this case it's $1\!=\!1$, and its falsifying literal L' is $1\!=\!2$.

We call the `resolve` function with c11, L, and L''s reason, which is c12, and perform the resolution returning a resolved clause. Below is how the resolution is done:

c11 : $\{1\!=\!1, 3\!=\!0, 2\!=\!1\}$

c12 : $\{1\!=\!2, 4\!=\!1, 5\!\neq\!1\}$

resolved clause then is

Res : $\{3\!=\!0, 2\!=\!1, 4\!=\!1, 5\!\neq\!1\}$

We call `AnalyzeConflict(`C`)` again, passing the resolved clause as an argument this time. The resolved clause does not meet the `Potent` function criteria, since there are again two literals in the resolved clause that were falsified at the current level, so we pick from it a literal whose falsifier has maximal index. It's $4\!=\!1$ and the reason for it to be false is literal $4\!=\!3$. We perform resolution on the resolved clause and the reason for $4\!=\!3$, which is c14, and call `AnalyzeConflict(`C`)` again.

Res : $\{3\!=\!0, 2\!=\!1, 4\!=\!1, 5\!\neq\!1\}$

c14 : $\{4\!=\!3, 6\!=\!2, 2\!=\!0\}$

new resolved clause is

$\{3\!=\!0, 2\!=\!1, 5\!\neq\!1, 6\!=\!2, 2\!=\!0\}$

Still the `Potent` function criterion is not met, since the resolved clause again has two literals that were falsified at the current level. (Recall that the resolved clause is learned clause if it has exactly one literal falsified at the current level.) So we perform yet another round of `AnalyzeConflict(`C`)`. L is $3\!=\!0$, L' is $3\!=\!1$ and the reason for L' is c13.

Res : $\{3\!=\!0, 2\!=\!1, 5\!\neq\!1, 6\!=\!2, 2\!=\!0\}$

c13 : $\{3\!=\!1, 6\!=\!2, 5\!=\!0\}$

new resolved clause is

$\{2\!=\!1, 5\!\neq\!1, 6\!=\!2, 2\!=\!0, 5\!=\!0\}$

This time the resolved clause satisfies the `Potent` function: the only literal falsified at the current level is $6=2$. Now the learned clause is

$$\{2=1,\, 5\neq1,\, 6=2,\, 2=0,\, 5=0\}$$

So the resolved clause is the learned clause.

Add the learned clause to the theory and compute the backtrack level. The learned clause is passed to `getBacktrackLevel(`C`)` function which computes the level the search should return to. In this case the level is 3 since the current level is 5, and 3 is the maximal level (of falsification) from the learned clause which is not the current level. Literals $5\neq1$ and $5=0$ from the learned clause were falsified at level 3. So we backtrack to level 3.

$$P_{index} = \{2=0_0,\, 7=2_1,\, 9=0_2,\, 10=1_3,\, 11=2_4,\, 5=1_5\ \}$$
$$P_{level} = \{2=0_0,\, 7=2_1,\, 9=0_1,\, 10=1_2,\, 11=2_2,\, 5=1_3\ \}$$
$$P_{reason} = \{2=0_{c0},\, 7=2_D,\, 9=0_{c1},\, 10=1_D,\, 11=2_{c2},\, 5=1_D\}$$

The learned clause is now unit, with unit literal $6=2$.

3   Implementation

This solver is written in C++ using data structures such as vectors and lists provided by the Standard Template Library (STL) as well as custom data structures built by us. In this section we present the details of our data structures, show how they are related and how they operate, and also present the details of procedures used in our implementation of the Extended DPLL algorithm described in the previous chapter.

3.1   Data Structures

The solver is built with 4 main classes: Literal, Clause, Variable, and Formula, and a Structure VAR-RECORD. Literal is the simplest, most basic one, while Formula is the most complex and functional one.

3.1.1   Literal

This class has data members for each of the following: `VAR(int)`, `VAL(int)`, and `EQUAL(bool)`. `VAR` represents the literal's variable, `VAL` represents the literal's value, and `EQUAL` represents the positive literal if it is **true**, negative literal if it is **false**.

3.1.2   Clause

This class contains a vector of Literals, `ATOM_LIST`; a data member `SAT` which is **true** if the clause is satisfied, otherwise **false**; `NumAtom` which represents number of literals in clause; `NumUnAss` which represents number of literals not yet assigned; and a `LEVEL` which is some integer greater than or equal to 0 representing the depth in the search space when this clause was satisfied.

3.1.3   VARRECORD

This is a simple structure of linked list used to link clauses. It has two data members: `c_num(int)` which points to the clause number, and `NEXT(VARRRECORD *)` which points to the next record.

3.1.4 Variable

As the name suggests, this class represents the variables in the theory. It has the following data members: `VAR` representing the variable; `DOMAINSIZE` representing the size of the domain of `VAR`; `VAL` representing the value assigned to `VAR`; `SAT` is **true** if variable is assigned, otherwise **false**; `LEVEL` representing the depth of search space when the variable was assigned. Since a literal can be positive (i.e. of the form $V{=}x$) or negative (i.e. of the form $V{\neq}x$), it has arrays storing literal counts for each of them, (i.e. `ATOMCNTPOS`, `ATOMCNTNEG`) where each index represents a domain value; records of clauses in which each literal occurs (i.e. `ATOMRECPOS`, `ATOMRECNEG`); level at which each literal was assigned (i.e. `ATOMLEVEL`); and the reason for each literal to be assigned (i.e. `CLAUSEID`).

`ATOMCNTPOS`, `ATOMCNTNEG` are bookkeeping records that are used for branching heuristics. `ATOMRECPOS`, `ATOMRECNEG` are pointers to clauses which are used when traversing through the clause database, and `CLAUSEID` is used in conflict analysis to find the clause which caused the literal to be satisfied/falsified.

There is also another array `ATOMASSIGNED` whose index value is **true** if the positive literal is assigned, and **false** if negative literal is assigned.

3.1.5 Formula

This class is the main class; it represents the theory. It has two vectors: `VARLIST`, which represents all the variables in the theory, and `CLAUSELIST`, which stores all the clauses in the theory. It also has some auxiliary data members which keep track of number of backtracks that have occurred, number of decisions made, number of unit propagations done, etc.

This class implements the Extended DPLL algorithm and contains all the necessary functions required by the algorithm.

Below is a brief description of how the objects are initialized from a problem file (the file is in CNF format).

As we read the problem file, for every variable we read we create a Variable object, set it to VAR, and initialize the arrays with the domain size we read setting the default values. For every clause we read, we

create Literal objects and push them into the clause incrementing the number of literals NumAtom, and setting default values for the clause. We also update the Variable object of that literal by linking the clause to it, incrementing the atom count depending on the type of literal (i.e. positive or negative). In this way the Variables object knows about every clause it occurs in and the clause also knows what Variables occur in it, therefore getting a two-way relationship.

3.2   Procedures

checkSat() - This procedure checks if the theory is empty or not by visiting each clause in the theory and checking if SAT is true or not. It returns true if the theory is empty and false otherwise.

checkConflict() - An empty clause is a clause which has 0 unassigned literals and is unsatisfied (i.e. SAT is false). This procedure checks if there is an empty clause in the theory. It returns the empty clause number when conflict, -1 otherwise.

checkEntail() - This procedure checks for entailment of an atom by the current partial interpretation P. It visits the variable and checks if all but one domain value has a negative literal in the current P. If there is one then this is an entailed atom return it, null otherwise.

chooseLiteral() - This procedure chooses the next branching literal using a simple heuristic whereby we pick a literal which is not yet satisfied, and which satisfies maximum clauses and removes minimum literals from unsatisfied clauses. We utilize the literal counts here.

reduceTheory() - This procedure takes in a literal as an argument and updates the theory such that i)if the literal is a positive literal, mark all unsatisfied clauses in which the literal occurs as satisfied (i.e. SAT = true), and reduce the literal count of all the clauses in which the negative literal occurs. ii)if the literal is a negative literal, mark all unsatisfied clauses in which the literal occurs as satisfied (i.e. SAT = true), and reduce the literal count of all the clauses in which the positive literal occurs.

satisfyClause() - This procedure is called from the reduceTheory() procedure and it takes a literal as an argument. It satisfies unsatisfied clauses where the literal occurs and updates the literal counts of literals that occur in that clause.

removeLiteral() - This procedure is called from the reduceTheory() procedure and it takes a literal as an
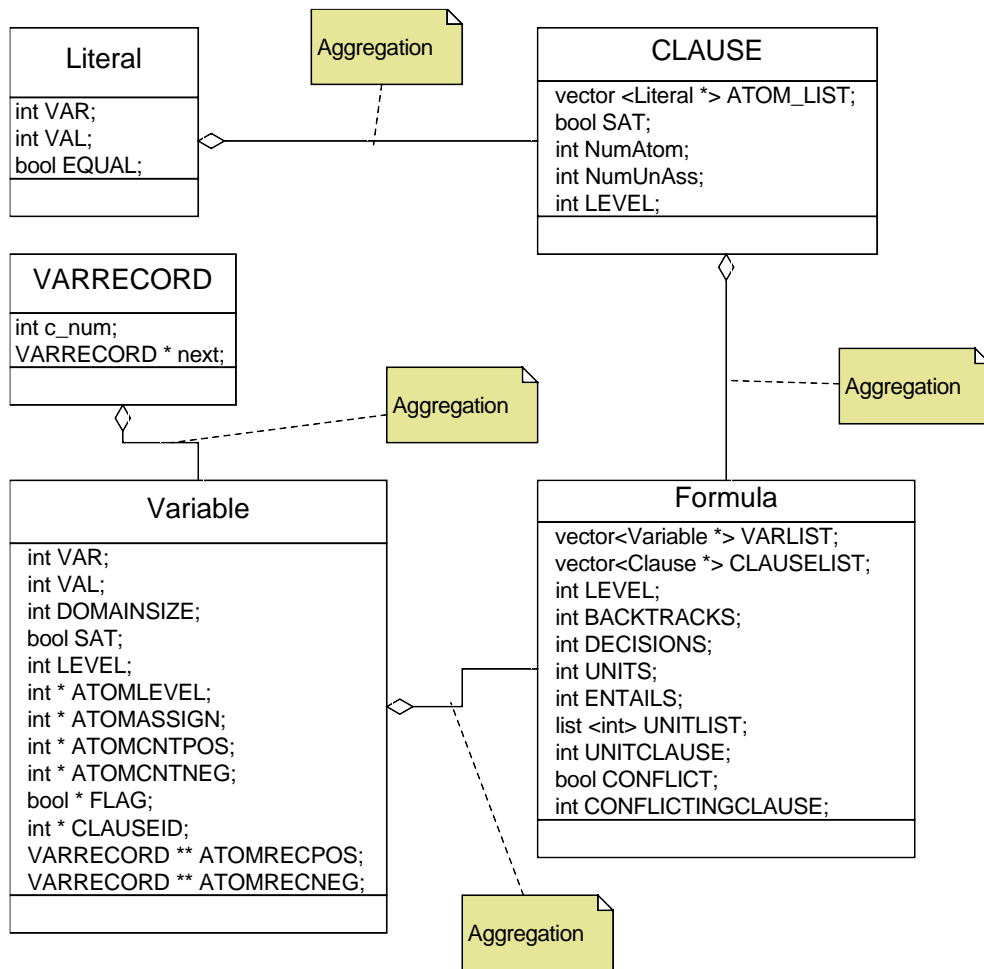
Figure 4: Class Diagram

27

argument. It reduces the number of unassigned literals in the clause by 1. It also checks if the number of unassigned literals is 1, in which case the clause is unit and it adds this clause to UNIT_LIST. If it's 0, then we have a conflict and it flags the procedure and immediately stops proceeding forward. The analyzeConflict procedure is called to handle the conflict.

undoTheory() - This procedure takes in a level argument and unmarks all the clauses and literals that were marked before this level. The LEVEL data member of Variable and Clause class are used to determine whether to unmark or not.

analyzeConflict() - This procedure is called when a conflict (i.e. an empty/conflicting clause) is detected in the theory. It takes the conflicting clause, finds the latest literal which got falsified at the current level, resolves the conflicting clause with the reason for the literal. It does the above process till the resolvent contains only 1 literal which was falsified at the current level. The resolvent clause is now our learned clause, which we add to the theory. The backtrack level is returned, which is the latest level other than the current level in which a literal from the learned clause was falsified.

unitPropagation() - This procedure takes a unit clause from the UNIT_LIST, finds the unit literal and satisfies it, thus satisfying the clause. While there is a unit clause in UNIT_LIST, it performs unit propagation.

NonChronoBacktrack() - This is the main procedure which performs the Extended DPLL algorithm. It has an infinite while loop which stops only when either the problem is found to be satisfiable, or unsatisfiable, or the time limit for searching is reached. It starts by first checking if the theory is satisfied or not, if it is satisfied then it returns satisfiable and exits the loop. It then checks for unit clauses in the theory and satisfies them. After this step it checks for any conflicts in the theory. If there is one then it will analyze the conflict, learn the conflict clause, add it to the theory, and backtrack. If the conflict occurred at level 0 then the problem is unsatisfiable and unsatisfiable is returned. If all goes well till this step then it's time to make a decision (i.e. choose a branching literal) and start the loop again.

4 Evaluation Procedure and Experimental Results

To test our Finite Domain SAT solver we need Finite Domain problems. One source for these is a random Finite Domain problem generator. This generator generates random satisfiable and unsatisfiable problems depending on input criteria such as number of variables, size of domain, number of clauses, and size of clauses.

To generate a satisfiable problem, we first generate a random interpretation of the signature with the number of variables defined by user. Then for each clause we pick randomly one literal satisfied by this interpretation and then fill in the clause with other literals (positive/negative which are distinct from this literal) randomly. By following this method we are guaranteed to have at least one model of the problem. Although it makes some instances easy to solve, we have observed quite a variation in difficulty as the problem size grows.

To generate a random unsatisfiable problem, the process is similar as above except we don't generate a model. For each clause we randomly pick a literal (positive or negative) till the clause size is reached, and then we start with another clause. This does not guarantee that the problem is unsatisfiable though. We have observed that when the number of clause to number of variable ratio is between 1 and 2 most of the problems generated are satisfiable. To see if the problem is unsatisfiable we run it or its Boolean conversion.

Alam [1] in his thesis this year has developed additional Finite Domain benchmark problems. We have also used these problems in our evaluation and report their results. Here we briefly describe the problems but for more details the reader should read his work. The three problem sets that we have used are GT_N, Pebbling, and Pigeon Hole.

GT_N formulas are unsatisfiable formulas that encode the false claim that there exists a partial order of a Finite set N that does not have a maximal element. Pebbling formulas are based on so-called pebbling graphs; they are minimally unsatisfiable, that is, they can be made satisfiable by randomly removing a single clause. Pigeon hole formulas encode a very common NP-complete problem with the objective of putting $N + 1$ pigeons in N holes with a constraint of one pigeon per hole.

4.1 Syntax : Finite Domain Problems

The Finite Domain problem file uses syntax introduced by Nagle [17] which is an extension of the standard DIMACS CNF format commonly used for Boolean SAT problems. The problem file contains all the information needed to describe the problem. It has two sections : the preamble and the body.

The preamble contains comments, information about the size of the problem, number of variables, and the domain size of each variable. Each line in the preamble starts with a character which determines the property of the line. There are 3 characters used : {c, p, d}.

1. Comment line: This line contains comments and can be ignored.

   c This is a comment line

2. Problem line: This line contains information about the problem. It begins with a p. There is exactly one such line for each problem.

   p cnf <NumVar> <NumClause>

   where NumVar is the total number of variables in the problem, and NumClause is the number of clauses in the problem.

3. Domain line: This line contains information about the domain size of a variable. It begins with a d and is followed by the variable and then by the domain size.

   d <VarName> <DomainSize>

   where VarName is the variable name, and DomainSize is the size of the domain of the variable.

There should be at most one domain life for each variable. If a variable has no domain line, it is assumed to be Boolean (domain size 2).

Although a comment line can occur anywhere in the file, we prefer it to occur at the beginning, followed by the problem line, then all the domain lines, and then the clauses. Each clause ends with a 0, which is used

as an endmarker, and the variables are represented by numbers from 1 to N where N is the total number of variables in the theory.

Figure 5 shows a sample Finite Domain problem file displaying the preamble and body.

```
c This is a sample Finite Domain problem file
c randomly generated by the generator
c It has 4 variables and 16 clauses
c This problem is satisfiable and has a model
c {1=1, 2=4, 3=2, 4=3}
p cnf 4 16
d 1 5
d 2 5
d 3 5
d 4 5
1=1 2=1 3=1 0
4=3 3=0 1=0 0
3=2 4=4 2=2 0
2=4 1=0 3=0 0
4=3 3=3 1=4 0
1=3 2=4 3=4 0
4=3 3=3 2=3 0
1=1 2=0 3=0 0
4=3 3=2 1=1 0
3=2 4=2 2=1 0
2=4 1=1 3=1 0
4=2 3=2 1=2 0
1=3 2=4 3=1 0
1=0 2=0 3!=0 0
4=4 3=2 2!=3 0
3=3 1=1 4!=3 0
```

Figure 5: A Finite Domain problem file

32

Let us look at the last line from the sample file, `3=3 1=1 4!=3 0`. 0 is the delimiter. The line represents a clause with 3 literals. The string 4!=3 represents the literal $4 \neq 3$. (We use != to represent $\neq$).

4.2   Evaluation Procedure

To evaluate the solvers, we randomly generated Finite Domain problems, varying numbers of variables, domain size, clause size, and problem size. We used the following combinations to generate the problem.

Variables Size : $\{100, 200, 300, 500, 700, 1000, 1500\}$

Domain Size   : $\{2, 3, 5\}$

Clause Size   : $\{3, 5\}$

Clause Ratio   : $\{1, 2, 3, 5, 7, 10\}$

Clause Ratio stands for the (number of clauses / number of variables) in the problem.

So for example one combination would have 100 variables, domain size 2, clause size 3, and the clause ratio 1; another would have 100 variables, domain size 2, clause size 3, and the clause ratio 2. In total we have 252 combinations. We generated 3 satisfiable and 3 unsatisfiable problems for each of the combinations, getting 1512 Finite Domain problems. Due to space limitations we will not report all the results in detail; we will report a few interesting ones and then generalize our observations.

Besides the above test set, we also generated the following test sets which are structured problems : GT_N(11 problems), Pebbling(24 problems), and Pigeon Hole (10 problems). Each instance has different number of variables and clauses.

In order to use the above test sets for comparison between Finite Domain solvers and Boolean solvers we needed to encoded these problems into Boolean SAT problems. We use Linear Encoding as our technique to do the encoding. Sinha [20] reported that `zchaff`, the Boolean solver which we use for these comparisons, performs better when used Linear Encoding then Quadratic Encoding. These encodings are specified in Appendix A. We also needed to transform these test sets into `CAMA`'s input format because of difference in representation. (This transformation is straightforward. $V = x$ becomes $V^{\{x\}}$ and $V \neq x$ becomes $V^{\{x' | x' \in dom(V), x' \neq x\}}$.)

Most SAT solvers' results are reported as search time and/or run time. Search time is the time spent searching the solution search space while run time is the sum of search time and time spent reading the problem and initializing the data structures. We use both measures, along with number of backtracks and number of decisions made, in our evaluation.

We use following solvers to compare our solver with: `zchaff` [16], `CAMA` Finite Domain solver [13], Sinha's Finite Domain solver [20] (hereon Sinha), and Nagle's Finite Domain solver [17] (hereon Nagle). Sinha and Nagle both do chronological backtracking, so to compare we also have our own chronological backtracking solver. From here on we will refer to our solver with chronological backtracking as `FDS_CH`, and our solver with non-chronological backtracking and conflict analysis as `FDS_NC`.

We perform the following comparisons:

1. Comparison of Sinha, Nagle, `FDS_CH`, and `FDS_NC`

2. Comparison of `FDS_CH` and `FDS_NC`

3. Comparison of `CAMA` and `FDS_NC`

4. Comparison of `zchaff` and `FDS_NC`

4.2.1 Comparison with Sinha, and Nagle

The aim of this experiment is to show that our solvers (`FDS_CH`, `FDS_NC`) perform better than Sinha and Nagle. Sinha has limitation in the input language such that it does not accept negative literals. Therefore we test it with either problems with no negative literals or convert problems with negative literals into ones without negative literals.

On the randomly generated test data (i.e. 1512 problems), `FDS_CH` outperforms Sinha 62% times, Nagle 68% times; `FDS_NC` outperforms Sinha 87% times, Nagle 93% times. Although these results are good, given smaller problems where the Clause Ratio is between 1 and 2, Sinha performs slightly better than `FDS_CH`, and `FDS_NC`.

Another way of looking at the results is by comparing number of problems solved. `FDS_NC` solved all the problems in the test data, while `FDS_CH` times out on few. Sinha's solver timed out on problems more than

`FDS_CH` did, and Nagle's solver timed out on problems more than Sinha's solver did. The time out limit was 600 seconds.

Table 1 shows the results of search time for Sinha, Nagle, `FDS_CH`, `FDS_NC`, while Figure 6 displays the graph for problems of variable size 500. All problems are satisfiable.

Table 2 shows the results of search time for Sinha, Nagle, `FDS_CH`, `FDS_NC` for problems of variable size 200. All problems are unsatisfiable.

Clauses	Sinha	Nagle	FDS_CH	FDS_NC
500	0.0102	0.02358	0.0208	0.0265
1000	0.0357	0.0390	0.0419	0.0326
1500	0.0350	0.0446	0.0596	0.0277
2500	0.0613	0.0811	0.0240	0.0215
3500	0.2187	0.1416	0.0415	0.0172
5000	0.5098	0.2998	0.0363	0.0177

Table 1: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC. Number of Variable = 500, Clause size = 3, Domain size = 3. All problems are satisfiable.

Clauses	Sinha	Nagle	FDS_CH	FDS_NC
1000	31.6345	217.538	15.6542	0.0151
1400	4.8109	8.0288	1.4816	0.0136
2000	0.9819	1.3373	0.1327	0.0147

Table 2: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC. Number of Variable = 200, Clause size = 3, Domain size = 3. All problems are unsatisfiable.

As we can see from the above table FDS(CH, NC) performs better overall than Sinha, and Nagle on randomly generated problems. The next three tables shows a similar pattern on structured problems.

Note: The name of the files in the below tables are in a format which tells us how many variables, clauses, domain sizes, etc are in the problem. It is an easy way to represent the information. For example v10_c152_d3_cs3_s1 means 10 variables, 152 clauses, domain size of 3, clause size is 3, and it is satisfiable. If it is s0 then the problem is unsatisfiable. Sometimes we mention some common information to all the files in the caption.
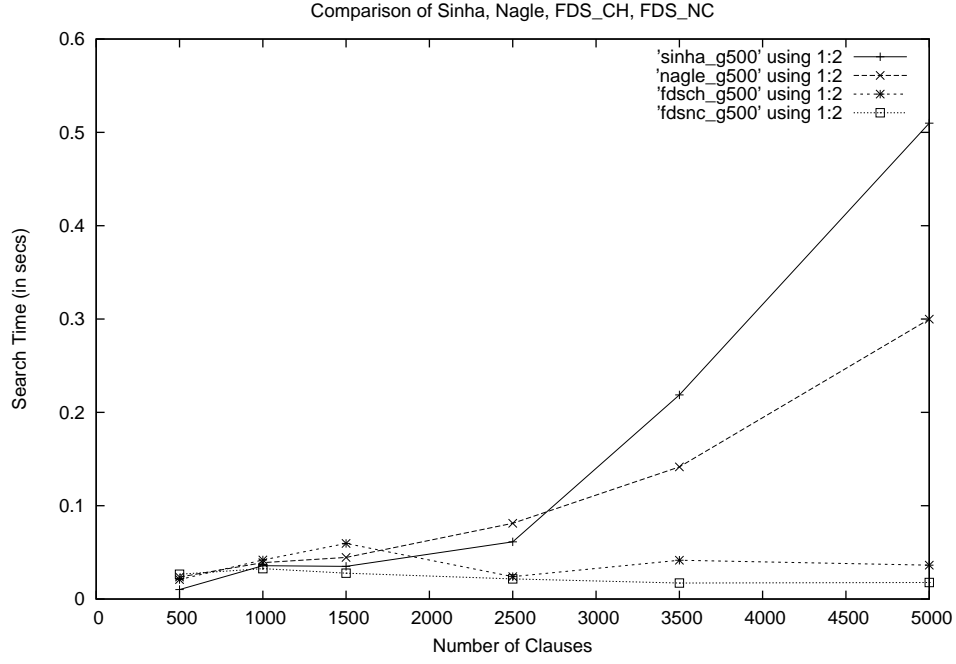
Figure 6: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC. Number of Variable = 500, Clause size = 3,

Domain size = 3. All problems are satisfiable.

File Name	Sinha	Nagle	FDS_CH	FDS_NC
v6_c75	0.07	0.01	0.00	0.00
v7_c126	1.31	0.16	0.02	0.01
v8_c196	26.05	1.83	0.14	0.02
v9_c288	513.13	17.39	1.35	0.03
v10_c405	600	266.77	17.80	0.05
v12_c726	600	600	600	0.13
v15_1470	600	600	600	0.46

Table 3: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC on Pigeon hole formulas (which are all unsat-

isfiable), Domain size = 2

37

File Name	Type	Sinha	Nagle	FDS_CH	FDS_NC
v7_c251_d3_s1	SAT	0.00	0.00	0.00	0.00
v7_c252_d3_s0	UNSAT	0.01	0.00	0.00	0.00
v7_c1032_d4_s1	SAT	0.01	0.00	0.00	0.00
v7_c1033_d4_s0	UNSAT	0.51	0.02	0.02	0.02
v7_c3133_d5_s1	SAT	600	0.01	0.01	0.00
v7_c3134_d5_s0	UNSAT	600	3.01	0.30	0.14
v8_c738_d3_s1	SAT	0.00	0.00	0.00	0.00
v8_c739_d3_s0	UNSAT	0.16	0.05	0.02	0.02
v8_c4105_d4_s1	SAT	600	0.02	0.02	0.00
v8_c4106_d4_s0	UNSAT	600	11.22	0.97	0.21
v8_c15634_d5_s1	SAT	600	0.11	0.08	0.02
v8_c15635_d5_s0	UNSAT	600	156.44	11.08	1.14
v9_c2197_d3_s1	SAT	0.02	0.01	0.01	0.00
v9_c2198_d3_s0	UNSAT	11.30	1.89	0.32	0.08
v9_c16394_d4_s1	SAT	600	0.11	0.09	0.02
v9_c16395_d4_s0	UNSAT	600	219.68	18.27	1.15
v9_c78135_d5_s1	SAT	600	0.60	0.46	0.08
v9_c78136_d5_s0	UNSAT	600	600	246.09	7.07
v10_c6572_d3_s1	SAT	0.08	0.06	0.04	0.01
v10_c6573_d3_s0	UNSAT	136.3	600	4.46	0.39
v10_c65547_d4_s1	SAT	600	0.35	0.43	0.06
v10_c65548_d4_s0	UNSAT	600	600	285.33	5.51

Table 4: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC on Pebbling formulas

File Name	Sinha	Nagle	FDS_CH	FDS_NC
v20_c75_d2_s0	0.00	0.00	0.00	0.00
v30_c141_d2_s0	0.00	0.01	0.00	0.00
v42_c238_d2_s0	0.01	0.22	0.00	0.00
v56_c372_d2_s0	0.02	2.77	0.01	0.00
v72_c549_d2_s0	0.05	64.18	0.04	0.01
v90_c775_d2_s0	0.13	600	0.09	0.01
v110_c1056_d2_s0	0.38	600	0.23	0.02
v156_c1807_d2_s0	3.53	600	1.35	0.03
v210_c2850_d2_s0	65.26	600	7.46	0.06
v272_c4233_d2_s0	600	600	40.53	0.10
v380_c7050_d2_s0	600	600	550.85	0.22

Table 5: Comparison of Sinha, Nagle, FDS_CH, and FDS_NC on GT_N formulas. All formulas are unsatisfiable.

4.2.2   Comparison of FDS_CH and FDS_NC

The aim of this set of experiments is to see if clause learning (conflict analysis) and non-chronological back-tracking helps us in solving problems. We compare the search time and number of backtracks done on the same set of problems by the two solvers FDS_CH and FDS_NC. FDS_CH implements chronological back-tracking, while FDS_NC implements clause learning and non-chronological backtracking. The algorithm for chronological backtracking is same as one Nagle [17] implemented in his solver, excluding the pure literal condition. The reason we implemented FDS_CH was to fairly compare a solver with FDS_NC which does not have learning. We observe that due to data structures and implementation variations neither Nagle's [17] nor Sinha's [20] solver was helpful because even without clause learning and non-chronological back-tracking our solver outperform Nagle's and Sinha's solvers. Also for the FDS solvers, everything except the Extended DPLL function is the same, which gives a fair comparison.

Looking at the search time of the two solvers from Table 3, Table 4, and Table 5 we see that FDS_NC almost always outperforms FDS_CH.

Table 6, Table 7, and Table 8 show the number of backtracks done by each solver. We only display results from UNSAT problems because most of the time for SAT problems we have observed 0 backtracks.

Observing the number of backtracks for each set of problems we see that FDS_NC has much fewer backtracks than FDS_CH. It appears that as the problem size grows the growth in number of backtracks in FDS_NC is linear while in FDS_CH it looks exponential.

Similar results are obtained on the randomly generated problems. Almost 99% of the time FDS_NC outper-forms FDS_CH.

From the number of backtracks and search time comparisons we can now conclude that learning from the conflicts helps in solving the problem.

4.2.3   Comparison with CAMA

CAMA [13] is a Finite Domain solver that was developed in 2003 by C. Liu and group. This is the other Finite Domain solver which incorporates clause learning that we know about. It is publicly available. Our

File Name	FDS_CH	FDS_NC
v6_c75	31	11
v7_c126	141	16
v8_c196	870	22
v9_c288	6955	29
v10_c405	73313	37
v12_c726	2102839*	56
v15_1470	1018193*	92

Table 6: Comparison of number of backtracks done by FDS_CH, and FDS_NC on Pigeon hole formulas (which are all unsatisfiable), Domain size = 2. (* = timed out)
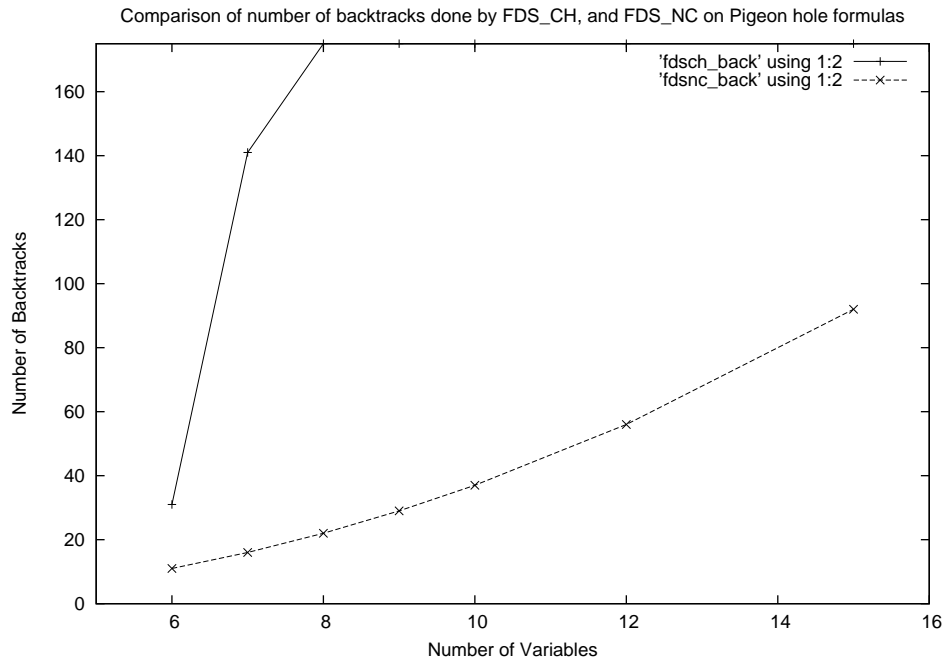
Figure 7: Comparison of number of backtracks done by FDS_CH, and FDS_NC on Pigeon hole formulas (which are all unsatisfiable), Domain size = 2. (* = timed out). For plotting purposes maximum backtracks shown is 175.

41

File Name	FDS_CH	FDS_NC
v7_c252_d3_s0	26	9
v7_c1033_d4_s0	74	14
v7_c3134_d5_s0	152	19
v8_c739_d3_s0	74	11
v8_c4106_d4_s0	287	17
v8_c15635_d5_s0	778	23
v9_c2198_d3_s0	208	13
v9_c16395_d4_s0	1095	20
v9_c78136_d5_s0	3516	27
v10_c6573_d3_s0	580	15
v10_c65548_d4_s0	4456	23

Table 7: Comparison of number of backtracks done by FDS_CH, and FDS_NC on Pebbling formulas.

File Name	FDS_CH	FDS_NC
v20_c75_d2_s0	4	4
v30_c141_d2_s0	8	5
v42_c238_d2_s0	16	6
v56_c372_d2_s0	32	7
v72_c549_d2_s0	64	8
v90_c775_d2_s0	128	9
v110_c1056_d2_s0	256	10
v156_c1807_d2_s0	1024	12
v210_c2850_d2_s0	4096	14
v272_c4233_d2_s0	16384	16
v380_c7050_d2_s0	131072	19

Table 8: Comparison of number of backtracks done by FDS_CH, and FDS_NC on GT_N formulas which are all unsatisfiable.

File Name	Search Time		# of Decisions		# of Backtracks	
	CAMA	FDS_NC	CAMA	FDS_NC	CAMA	FDS_NC
v6_c75	0.01	0.00	49	33	49	11
v7_c126	0.02	0.01	129	59	129	16
v8_c196	0.12	0.02	321	96	321	22
v9_c288	0.61	0.03	769	146	769	29
v10_c405	6.46	0.05	1793	211	1793	37
v12_c726	600*	0.13	-	394	-	56
v15_1470	600*	0.46	-	831	-	92

Table 9: Comparison of CAMA and FDS_NC on Pigeon Hole Formulas.(* = time out, 600 seconds)

aim here is to compare the performance of our solver with clause learning to CAMA and we'll use the search time, number of backtracks and decisions to evaluate both the solvers.

The Table 9, Table 10, and Table 11 display the search time, number of backtracks, and number of decisions made for the problems Pigeon Hole, Pebbling, and GT_N.

Figure 8 displays the graph for the search time for CAMA and FDS_NC on Pebbling formulas with domain size 4.

From these tables we can observe that FDS_NC outperforms CAMA in almost all problems. Looking at the number of decisions made, in the first two problem sets CAMA makes exponentially more branching decisions than FDS_NC but in the third problem FDS_NC makes more branching decisions than CAMA. The number of backtracks done by CAMA is always more than number of backtracks done by FDS_NC. But intuitively, the learning mechanism that CAMA has is similar to one FDS_NC has. The major difference between these two solvers is that CAMA uses a two-watched-literal scheme compared to FDS_NC's better dynamic statistics for heuristics. Discrepancy in number of backtracks is presumably due to this difference. It appears that, to the extent that CAMA remains competitive on search time, this is because weaker heuristic is somewhat offset by quicker backtracking (due to two-watched-literal scheme).

File Name	Search Time		# of Decisions		# of Backtracks	
	CAMA	FDS_NC	CAMA	FDS_NC	CAMA	FDS_NC
v7_c252_d3_s0	0.00	0.00	82	24	82	9
v7_c1033_d4_s0	0.06	0.02	257	38	257	14
v7_c3134_d5_s0	0.34	0.14	626	52	626	19
v8_c739_d3_s0	0.04	0.02	244	35	244	11
v8_c4106_d4_s0	0.41	0.21	1025	55	1025	17
v8_c15635_d5_s0	3.06	1.14	3126	75	3126	23
v9_c2198_d3_s0	0.16	0.08	730	48	730	13
v9_c16395_d4_s0	2.68	1.15	4097	75	4097	20
v9_c78136_d5_s0	38.50	7.07	15626	102	15626	27
v10_c6573_d3_s0	0.68	0.39	2188	63	2188	15
v10_c65548_d4_s0	48.91	5.51	16385	98	16385	33

Table 10: Comparison of CAMA and FDS_NC on Pebbling Formulas. (* = time out, 600 seconds)

File Name	Search Time		# of Decisions		# of Backtracks	
	CAMA	FDS_NC	CAMA	FDS_NC	CAMA	FDS_NC
v20_c75_d2_s0	0.00	0.00	6	12	6	4
v30_c141_d2_s0	0.00	0.00	10	21	10	5
v42_c238_d2_s0	0.00	0.00	15	38	15	6
v56_c372_d2_s0	0.01	0.00	21	56	21	7
v72_c549_d2_s0	0.00	0.01	28	83	28	8
v90_c775_d2_s0	0.01	0.01	36	116	36	9
v110_c1056_d2_s0	0.02	0.02	45	158	45	10
v156_c1807_d2_s0	0.04	0.03	66	267	66	12
v210_c2850_d2_s0	0.07	0.06	91	415	91	14
v272_c4233_d2_s0	0.12	0.10	120	613	120	16
v380_c7050_d2_s0	0.27	0.22	171	1009	171	19

Table 11: Comparison of CAMA and FDS_NC on GT_N Formulas.

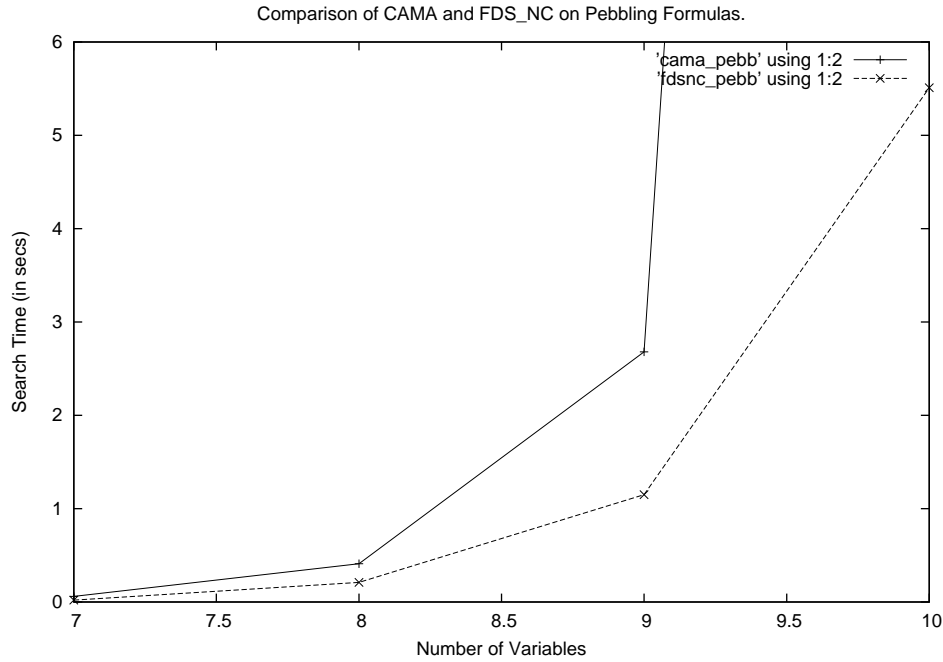Comparison of CAMA and FDS_NC on Pebbling Formulas.

Figure 8: Comparison of CAMA and FDS_NC on Pebbling Formulas. For plotting purposes Search Time limit has been kept 6 seconds. Domain size = 4

A point to note here from the tables is that the number of backtracks and number of decisions reported by CAMA are exactly the same on these problems. (On some other, randomly generated problems, this was not the case.) This seems quite unlikely to be correct, and suggests that there is probably a bug in CAMA's reporting mechanism.

The reported search times though seem to be reliable. And in general, when solution times climb steeply, it is because the solver is having to search more, in which case the number of backtracks and decisions will climb too.

Another point to note here is that CAMA performed extremely poorly when compared to FDS_NC on the randomly generated problem set (i.e. 1512 problems). FDS_NC performed better almost 95% of times.

47

File Name	Search Time		# of Backtracks	
	zchaff	FDS_NC	zchaff	FDS_NC
v6_c75	0.00	0.00	219	11
v7_c126	0.05	0.01	1032	16
v8_c196	0.60	0.02	5567	22
v9_c288	17.19	0.03	27969	29
v10_c405	239.90	0.05	89282	37
v12_c726	600*	0.13	-	56
v15_1470	600*	0.46	-	92

Table 12: Comparison of zchaff and FDS_NC on Pigeon Hole Formulas. (* = time out, 600 seconds)

4.2.4 Comparison with zchaff

zchaff is a state-of-the-art Boolean solver which is publicly available. We use this solver to test our FDS_NC with the goal in mind to show that Finite Domain problems are better solved using Finite Domain solver than encoding the problem in Boolean format and then solving using a Boolean solver.

As mentioned previously, we will use Linear Encoding to encode a Finite Domain problem into a Boolean Domain problem. This encoding is described in Appendix A.

File Name	Search Time		# of Backtracks	
	zchaff	FDS_NC	zchaff	FDS_NC
v7_c252_d3_s0	0.00	0.00	161	9
v7_c1033_d4_s0	0.02	0.02	767	14
v7_c3134_d5_s0	0.15	0.14	2499	19
v8_c739_d3_s0	0.01	0.02	485	11
v8_c4106_d4_s0	0.17	0.21	3071	17
v8_c15635_d5_s0	2.00	1.14	12499	23
v9_c2198_d3_s0	0.04	0.08	1457	13
v9_c16395_d4_s0	1.59	1.15	12287	20
v9_c78136_d5_s0	21.76	7.07	62499	27
v10_c6573_d3_s0	0.20	0.39	4373	15
v10_c65548_d4_s0	14.22	5.51	49151	33

Table 13: Comparison of zchaff and FDS_NC on Pebbling Formulas. (* = time out, 600 seconds)

File Name	Search Time		# of Backtracks	
	`zchaff`	`FDS_NC`	`zchaff`	`FDS_NC`
v20_c75_d2_s0	0.00	0.00	6	4
v30_c141_d2_s0	0.00	0.00	10	5
v42_c238_d2_s0	0.00	0.00	15	6
v56_c372_d2_s0	0.00	0.00	21	7
v72_c549_d2_s0	0.01	0.01	28	8
v90_c775_d2_s0	0.00	0.01	36	9
v110_c1056_d2_s0	0.00	0.02	45	10
v156_c1807_d2_s0	0.00	0.03	66	12
v210_c2850_d2_s0	0.00	0.06	91	14
v272_c4233_d2_s0	0.01	0.10	120	16
v380_c7050_d2_s0	0.01	0.22	171	19

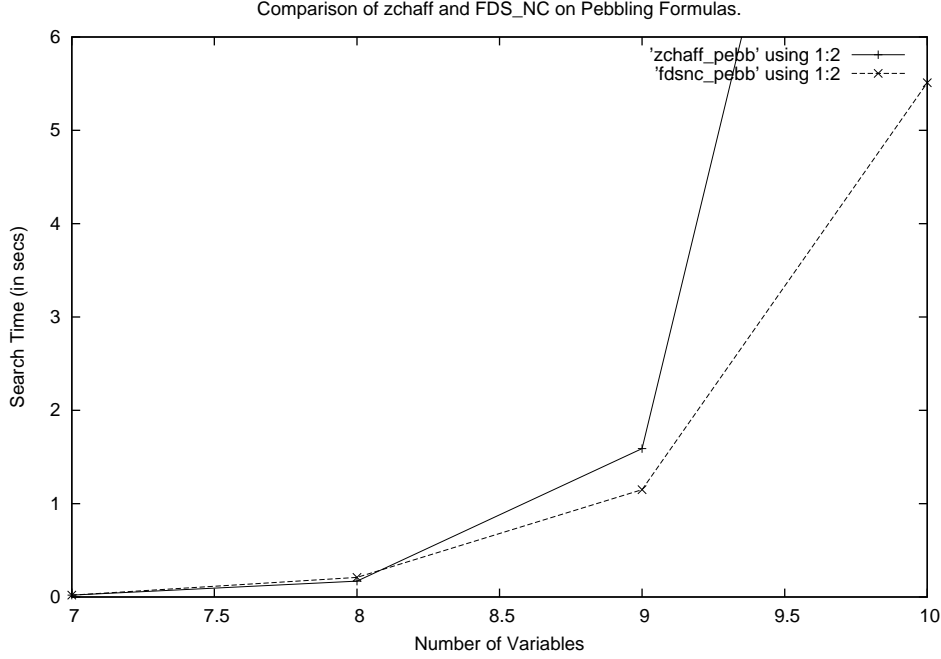Table 14: Comparison of `zchaff` and `FDS_NC` on GT_N Formulas.

Figure 9: Comparison of `zchaff` and FDS_NC on Pebbling Formulas. For plotting purposes Search Time limit has been kept 6 seconds. Domain size = 4

The Table 12, Table 13, and Table 14 displays the search time, and number of backtracks, made for the problems: Pigeon Hole, Pebbling, and GT_N.

We can observe from the result tables that `zchaff` and FDS_NC are competitive in search time. Looking at the number of backtracks, we see that `zchaff` does a lot of backtracking, while FDS_NC does relatively little. One reason for this is that when we encode the Finite Domain problem into a Boolean problem we add extra variables and clauses which might leads to extra conflicts and hence extra backtracking.

Although the FDS_NC is competitive with `zchaff` on structured problems, `zchaff` outperforms FDS_NC almost 66% of the time on randomly generated problems (i.e. 1512 problems). `zchaff` outperforms FDS_NC on almost all the satisfiable instances, while FDS_NC is competitive when it comes to unsatisfiable instances. Also as the problem grows larger and unsatisfiable FDS_NC starts to outperform `zchaff`.

One crucial difference between FDS_NC and `zchaff` that needs to be mentioned is the data structure. FDS_NC keeps track of each literal in the theory and dynamically updates the information related to the literal w.r.t. current partial interpretation. This leads to some extra work of bookkeeping and tracing. `zchaff`

avoids this extra work by using the two-watched-literal scheme where you watch two literals in a clause and visit the clause only if either one of the watched literals are affected. This gives `zchaff` some extreme speed, especially during backtracking, but reduces its capability for smart heuristics. As we saw in the comparison with `CAMA`, it appears that this tradeoff may favor `FDS_NC` as Finite Domain problems grow hard. This is a intriguing result, since up to now, the consensus seems to be that for Boolean solvers, `zchaff` is strongest overall.

4.2.5 Remarks on Comparison

Overall, the experiments conducted here are sufficient enough to tell us that work on Finite Domain solvers is important and necessary so as to take advantage of problems that can be nicely encoded in Finite Domain SAT. It is clear that there are many variations to try in clause learning and backtracking, in use of random restarts, in choice of heuristic. There are also still many opportunities for optimizing basic operations and data structures. We can also learn a great deal by experimenting with a wider range of real world problems encoded in Finite Domain SAT.

5 Conclusion

In this thesis we extended the Finite Domain DPLL algorithm introduced by Sinha [20], including the concept of negative literal introduced by Nagle [17]. The aim was to develop a solver that outperforms Sinha's, and Nagle's solvers and also be competitive with `CAMA` [13], a state-of-the-art Finite Domain solver, and `zchaff` [16], a state-of-the-art Boolean SAT solver. Sinha's and Nagle's solver do not learn when a conflict occurs, therefore we extended the Finite Domain algorithm by including clause learning and non-chronological backtracking.

We compared the performance of our solver with Sinha's solver, and Nagle's solver, and observed that our solver outperforms them on both random and structural problems which are satisfiable or unsatisfiable. Our solver also outperforms `CAMA` and is very competitive with `zchaff`, sometimes performing better than `zchaff`.

Although we have not reported on it in this thesis, we also compared the performance of various heuristics for selecting a decision literal, since a poor heuristic may increase solution times. The heuristic that gave the best performance is one which prefers the highest (*satisfaction - falsification*) value on a literal, where *satisfaction* means number of clauses satisfied when the decision literal is satisfied and *falsification* means the number of unsatisfied clauses whose number of unassigned literals is reduced when the decision literal is satisfied. This heuristic is the same as Nagle's heuristic 4.

Experimental results suggest that clause learning and non-chronological backtracking are very important to solve hard problems. For a Finite Domain solver to perform competitively with state-of-the-art Boolean SAT solver `zchaff`, it seems to be necessary for the solver to learn from conflict and backtrack accordingly.

Overall, this thesis provides further evidence that a Finite Domain problem can be efficiently and competitively solved using a Finite Domain SAT solver instead of encoding it into a Boolean SAT format and using a state-of-the-art Boolean SAT solver. It also provides evidence that clause learning and non-chronological backtracking are essential components in the algorithm.

5.1 Future Work

Improvements of Data Structures

Data Structures are one of the most important components of the solver. Although much research has been done on data structures for Boolean SAT solvers, there is not much done for Finite Domain SAT solvers. Extending and analyzing the data structures in various Boolean solvers might give us more ideas about how to implement data structures for Finite Domain solvers.

Clause Learning and Non-Chronological Backtracking

For a SAT solver to perform efficiently on hard problems, learning from conflicts is very important. A better or more efficient way of analyzing and learning from conflicts would speed up the search process.

Solve Real World Problems

Although in this thesis we have used three real world problems (Pigeon Hole, Pebbling, and GT_N formulas) for experiments and comparisons besides randomly generated Finite Domain problems, it would be nice to have more benchmark problems. In practice, with Boolean SAT solvers, it is often observed that performance on structured "real-world" problems is quite different from performance on random problems. It would also be nice to see if this observation holds as well for Finite Domain solvers.

Integration with CCalc

CCalc [12] is a 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 Finite Domain SAT solver could be an extension that is useful.

Random Restarts

Random restarts has become a common technique used in recent Boolean SAT solvers. It periodically empties the partial interpretation and starts all over again. This ensures that the learned clauses are utilized from the beginning and possibly leads to search in a new space in the space of partial interpretations.

Further work on heuristics

Heuristics play a crucial role in the search for a model. More work is needed to compare the tradeoffs between fast implementation and stronger statistics for heuristics.

Finite Domain problems

To test a Finite Domain solver we require Finite Domain problems. So far we have very limited sets of structured problems to test on. Having more problems would be helpful.

A Translation of Finite Domain Theories

In this appendix we describe two encoding techniques used to convert Finite Domain SAT problems into Boolean SAT problems.

The two encodings we discuss are Quadratic Encoding and Linear Encoding. These encodings are adapted from Appendix A of [9], and [4]. In our experiments, we used the Linear Encoding, but also present the Quadratic Encoding because it is simpler, and seeing it first can make the Linear Encoding easier to understand.

The last section in this appendix is on how to eliminate negations from Finite Domain problems so that these problems can be run on solvers that do not support negation.

A.1 Quadratic Encoding

This is a straightforward encoding whereby each atom from the Finite Domain theory is understood as a Boolean variable. A literal in Finite Domain expressed as $V = x$ is represented as Boolean SAT literal expressed as $V = x$, and $V \neq x$ is expressed as $\neg V = x$.

An interpretation in the Finite Domain setting maps a variable V to *exactly* one value from $dom(V)$. To maintain this property in the Boolean translation, additional clauses are added to the theory. For each variable V, we add the following clauses.

$$\{V = x \mid x \in dom(V)\} \tag{3}$$

$$\{\neg V = x, \ \neg V = y\} \qquad (x, y \in dom(V) \ and \ x \neq y) \tag{4}$$

Clause (3) enforces that "at least one" atom $V = x$, where $x \in dom(V)$ is satisfied for a variable V from the Finite Domain theory, and clause (4) enforces that "at most one" of these literals is satisfied.

The number of Boolean variables created is

$$\sum_{V} \mid dom(V) \mid$$

and the number of clauses is the sum of the clauses from original theory and the clauses introduced by (3) and (4). For each variable V in the Finite Domain theory, clause (4) introduces a quadratic number of clauses based on the cardinality of *dom(V)*, hence the name of the encoding.

To state the correctness of this translation let's use this preliminary definition.

For each interpretation I of the variables of Finite Domain theory T, let I_Q be the interpretation of the corresponding Boolean theory such that, for all variables V of the language of T and all values $x \in dom(V)$, $I(V) = x$ iff $I_Q(V = x) = t$

Fact: *Let T be a Finite Domain theory and T' be the Boolean theory obtained by Quadratic encoding. I is a model of T iff I_Q is a model of T'. Moreover, every model of T' can be written in the form of I_Q, for some interpretation I of T.*

An Example:

Let's use the Finite Domain theory

$$\{\{A=0, B=0\}, \{A \neq 0, B=1\}, \{B \neq 1, C=2\}, \{A=0, B=1, C=2\}\} \tag{5}$$

where there are three variables {A, B, C}, each having domain {0, 1, 2}. The translated theory is shown below.

$$\{A=0, B=0\}, \{\neg A=0, B=1\}, \{\neg B=1, C=2\}, \{A=0, B=1, C=2\} \tag{6}$$

$$\{A=0, A=1, A=2\}, \{B=0, B=1, B=2\}, \{C=0, C=1, C=2\} \tag{7}$$

$$\{\neg A=0, \neg A=1\}, \{\neg A=0, \neg A=2\}, \{\neg A=1, \neg A=2\} \tag{8}$$

$$\{\neg B=0, \neg B=1\}, \{\neg B=0, \neg B=2\}, \{\neg B=1, \neg B=2\} \tag{9}$$

$$\{\neg C=0, \neg C=1\}, \{\neg C=0, \neg C=2\}, \{\neg C=1, \neg C=2\} \tag{10}$$

Clauses (6) are the Boolean version of the original theory, clauses (7) enforce the "at least one" property, and clauses (8, 9, 10) enforce the "at most one" property. From the above example, we see that a Finite Domain theory with 3 variables and 4 clauses requires 9 Boolean variables and 16 clauses. As the domain sizes in a Finite Domain theory increase, the size of the corresponding Boolean theory significantly increases and may require additional time to solve.

Ansotegui et al. [4] call this encoding the Standard Mapping.

A.2 Linear Encoding

This is the encoding that we use in our experiments to convert a Finite Domain theory into a Boolean theory. In this encoding the number of additional clauses grows linearly. Thus the name, Linear Encoding.

We start by converting the original theory into Boolean theory, as in the Quadratic Encoding and also add clauses (3) to enforce the "at least one" property. To enforce the "at most one" property, instead of using (4), which is quadratic, we define another way, which is linear.

We introduce for each variable V of the Finite Domain signature, $|dom(V) - 1|$ extra Boolean variables which will be used to enforce the "at most one" property. Although this encoding increases the number of variables (and also complicates the implementation of the translator), Sinha [20] showed that problems converted using Linear Encoding solve faster than problems converted using Quadratic Encoding for zchaff, the Boolean SAT solver that we use for comparisons.

Below is a definition of how to add the new additional clauses.

Let a Finite Domain variable V have domain $dom(V) = \{x_1, x_2, ..., x_n\}$.
Let the extra Boolean variables be $V_1, V_2, V_3, ..., V_{n-1}$.
The additional clauses will then be

$$\{\neg V_1, v = x_1\}, \{V_1, \neg v = x_1\}, \{\neg V_1, \neg v = x_2\}$$

$$\{\neg V_2, V_1, v = x_2\}, \{V_2, \neg V_1\}, \{V_2, \neg v = x_2\}\{\neg V_2, \neg v = x_3\}$$

$$...$$

$$\{\neg V_{n-2}, V_{n-3}, v = x_{n-2}\}, \{V_{n-2}, \neg V_{n-3}\}, \{V_{n-2}, \neg v = x_{n-2}\}, \{\neg V_{n-2}, \neg v = x_{n-1}\}$$

$$\{\neg V_{n-1}, V_{n-2}, v = x_{n-1}\}, \{V_{n-1}, \neg V_{n-2}\}, \{V_{n-1}, \neg v = x_{n-1}\}, \{\neg V_{n-1}, \neg v = x_n\} \qquad (11)$$

To state the correctness of this translation let's use this preliminary definition.

For each interpretation I of the variables of the Finite Domain theory T, let I_L be the interpretation of the corresponding Boolean theory such that, for all variables V of the language of T and all values $x \in dom(V)$ where $dom(V) = (x_1, x_2, ..., x_n)$, $I_L(V = x_i) = \mathbf{t}$ iff $I(V) = x_i$, and if $I(V) = x_i$, then for all $j \in \{1, 2, ..., n\text{-}1\}$

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

2) $I_L(V_j) = \mathbf{t}$, if $j \geq i$

Fact: *Let T be a Finite Domain theory and T' be its Linear encoding. I is a model of T iff I_L is a model of T'. Moreover, every model of T' can be written in the form I_L, for some interpretation I of T*

An Example:

Let's use the Finite Domain theory (5) where there are three variables $\{A, B, C\}$ each having domain $\{0, 1, 2\}$.

Let the extra variables be A_1, A_2, B_1, B_2, C_1, C_2.

The translated theory will consist of clauses (6), clauses (7), along with the following :

$$\{\neg A_1, A = 0\}, \{A_1, \neg A = 0\}, \{\neg A_1, \neg A = 1\}$$

$$\{\neg A_2, A_1, A = 1\}, \{A_2, \neg A_1\}, \{A_2, \neg A = 1\}, \{\neg A_2, \neg A = 2\}$$

$$\{\neg B_1, B = 0\}, \{B_1, \neg B = 0\}, \{\neg B_1, \neg B = 1\}$$

$$\{\neg B_2, B_1, B = 1\}, \{B_2, \neg B_1\}, \{B_2, \neg B = 1\}, \{\neg B_2, \neg B = 2\}$$

$$\{\neg C_1, C = 0\}, \{C_1, \neg C = 0\}, \{\neg C_1, \neg C = 1\}$$

$$\{\neg C_2, C_1, C = 1\}, \{C_2, \neg C_1\}, \{C_2, \neg C = 1\}, \{\neg C_2, \neg C = 2\} \tag{12}$$

A.3   Eliminating Negation

Nagle [17] in his work provides experimental evidence that including negations in Finite Domain theories and directly solving them without removing negation is advantageous in many cases. In addition, as we have seen, negation is also introduced in the clauses that are learned via conflict analysis. Sinha's solver [20] does not support negation and to use this solver in our experiments it is necessary to provide an encoding for eliminating negations.

The encoding is straightforward. Below is pseudocode which defines the translation.

```
Replace each clause C ∈ T with positive(C) where positive(C)

is computed recursively as follows.

positive(C)

  If C contains a negative literal V ≠ x

    return positive(C ∪ {V=x':x'∈dom(V) and x'≠x} - {V≠x})

  else

    return C
```

The following fact states the correctness of this translation.

Fact: *Let T be a Finite Domain theory (possibly with negation) and T' be the corresponding translated Finite Domain theory without negation. T and T' have the same models.*

An Example:

Let's use the Finite Domain theory (5) with three variables {A, B, C}, each having domain {0, 1, 2}. The translated theory is shown below.

$$\{\{A=0, B=0\}, \{A=1, A=2, B=1\}, \{B=0, B=2, C=2\}, \{A=0, B=1, C=2\}\} \qquad (13)$$

B Solver : How to use

In this chapter we describe the features available in the solver and show examples of how to use them. The features are :

1. Randomly Generating Finite Domain problems

2. Convert Boolean to Finite Domain

3. Convert Finite Domain to Boolean : Linear Encoding

4. Convert Finite Domain to Boolean : Quadratic Encoding

5. Finite Domain Solver : Chronological Backtracking

6. Finite Domain Solver : Non-Chronological Backtracking

B.1 Randomly Generating Finite Domain problems

This feature allows the user to randomly generate Finite Domain problems based on the criteria passed as arguments to the solver. It can generate Satisfiable problems using a randomly generated model. It can also generate problems without using a model.

Below is the format of executing.

```
exe -genben -var <int> -clause <int> -clausesize <int> -sat <1/0> -domain
<int> -drand <1/0> -file <string>
where :
```

- exe :   * name of executable

- -genben :   * option stating create benchmark problem

- -var :   * number of variables in benchmark problem

- -clause : * number of clauses in benchmark problem

- -clausesize : number of atoms in each clause, [DEFAULT : 3]

- -sat : states whether this benchmark problem is generated using a model
  or not [possible value : 1/0] [default : 1/true]

- -domain : states the domain size of each variables, [DEFAULT : 10]

- -drand : states whether to assign domain size of each variable randomly
  [possible value : 1/0] [DEFAULT : 0/false]

- -bool : states whether the file is in boolean cnf or modified cnf

- -file : * name of the output file

* - required fields

B.2 Convert Boolean to Finite Domain

This feature allows the user to convert a problem that is in Boolean format to Finite Domain format.

Below is the format of executing.

exe -b2f -file <string> -model <string>

where :

- exe : * name of executable

- -b2f : * option stating to convert file

- -file : * name of the boolean file

- -model : * name of the finite file

* - required fields

B.3 Convert Finite Domain to Boolean : Linear Encoding

This feature allows the user to convert a Finite Domain problem into a Boolean format using Linear Encoding.

Below is the format of executing.

```
exe -linenc -file <string> -model <string>
```

where :

- exe : * name of executable

- -linenc : * option stating to convert file

- -file : * name of the boolean file

- -model : * name of the finite file

* - required fields

B.4 Convert Finite Domain to Boolean : Quadratic Encoding

This feature allows the user to convert a Finite Domain problem into a Boolean format using Quadratic Encoding.

Below is the format of executing.

```
exe -quadenc -file <string> -model <string>
```

where :

- exe :   * name of executable

- -quadenc :   * option stating to convert file

- -file :   * name of the boolean file

- -model :   * name of the finite file

* - required fields

B.5 Finite Domain Solver : Chronological Backtracking

This feature allows the user to solve a Finite Domain problem using the Chronological Backtracking method.

Below is the format of executing.

```
exe -solvech -var <int> -clause <int> -file <string> -time <int>
```

where :

- exe :   * name of executable

- -solvech :   * option stating to solve the finite domain problem

- -var :   * number of variables in benchmark problem

- -clause :   * number of clauses in benchmark problem

- -file :   * name of the output file

- -time :   amount of time allowed for solver to run

* - required fields

B.6   Finite Domain Solver : Non-Chronological Backtracking

This feature allows the user to solve a Finite Domain problem using the Non-Chronological Backtracking method with Clause Learning.

Below is the format of executing.

```
exe -solvenc -var <int> -clause <int> -file <string> -time <int>
```

```
where :
```

- `exe :   * name of executable`

- `-solvenc :   * option stating to solve the finite domain problem`

- `-var :   * number of variables in benchmark problem`

- `-clause :   * number of clauses in benchmark problem`

- `-file :   * name of the output file`

- `-time :   amount of time allowed for solver to run`

```
* - required fields
```

References

[1] A. Alam. "A Finite Domain Solver". Master's thesis, University of Minnesota Duluth, August 2005.

[2] F. A. Aloul and K. A. Sakallah. "An Experimental Evaluation of Conflict Diagnosis and Recursive Learning in Boolean Satisfiability". In *International Workshop on Logic Synthesis*, 2000.

[3] C. Ansotegui, J. Larrubia, C. M. Li, and F. Manya. "Mv-Satz: A SAT Solver for Many-Valued Clausal Forms". In *Fourth Internation Conference Journees de l'Informatique Messine*, September 2003.

[4] C. Ansotegui and F. Manya. "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)*, May 2004.

[5] L. Baptista, I. Lynce, and J. P. Marques-Silva. "Complete Search Restart Strategies for Satisfiability". In *Proceedings of the IJCAI Workshop on Stochastic Search Algorithms (IJCAI-SSA)*, August 2001.

[6] C.M.Li and Anbulagan. "Heuristics based on unit propagation for satisfiability problems". In *International Joint Conference on Artificial Intelligence*, pages 366–371, 1997.

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

[8] A. M. Frisch and T. J. Peugniez. "Solving Non-Boolean Satisfiability Problems with Stochastic Local Search". In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 2001.

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

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

[11] C. P. Gomes, B. Selman, and H. Kautz. "Boosting combinatorial search through randomization". In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, 1998.

[12] http://www.cs.utexas.edu/users/tag/cc/.

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

[14] I. Lynce, L. Baptista, and J. P. Marques-Silva. "Unrestricted Backtrack Search for Propositional Satisfiability". Technical Report RT/03/2001, INESC-ID, Technical University of Lisbon, Portugal, July 2001.

[15] I. Lynce and J. P. Marques-Silva. "Building State-of-the-Art SAT Solvers". Technical Report RT/02/2002, INESC-ID, Technical University of Lisbon, Portugal, April 2002.

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

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

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

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

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

[21] P. Stock. "Solving Non-Boolean Satisfiability Problems with the Davis-Putnam Method". Master's thesis, University of York, March 2000.

[22] H. Zhang. "SATO: An efficient propositional prover". In *National Conference on Automated Deduction*, pages 272–275, 1997.

[23] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. "Efficient Conflict Driven Learning in Boolean Satisfiability Solver". In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 279–285, November 2001.

[24] L. Zhang and S. Malik. "The Quest for Efficient Boolean Satisfiability Solvers". In *Proceedings of 8th International Conference on Computer Aided Deduction(CADE 2002)*, July 2002.