Fast Incremental Unit Propagation by Unifying Watched-literals and Local Repair

by

Shen Qu

B.S. Aeronautics and Astronautics
Massachusetts Institute of Technology, 2004

SUBMITTED TO THE DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN AERONAUTICS AND ASTRONAUTICS
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

SEPTEMBER 2006

Signature of Author: __________

Department of Aeronautics and Astronautics
August 25, 2006

Certified by: ______________________________

Brian C. Williams
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: ______________________________

Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Committee on Graduate Students

Fast Incremental Unit Propagation by Unifying Watched-literals and Local Repair

by

Shen Qu

Submitted to the Department of Aeronautics and Astronautics on August 25, 2006 in Partial Fulfillment of the Requirements of the Degree of Master of Science in Aeronautics and Astronautics

ABSTRACT

The propositional satisfiability problem has been studied extensively due to its theoretical significance and applicability to a variety of fields including diagnosis, autonomous control, circuit testing, and software verification. In these applications, satisfiability problem solvers are often used to solve a large number of problems that are essentially the same and only differ from each other by incremental alterations. Furthermore, unit propagation is a common component of satisfiability problem solvers that accounts for a considerable amount of the solvers' computation time. Given this knowledge, it is desirable to develop incremental unit propagation algorithms that can efficiently perform changes between similar theories. This thesis introduces two new incremental unit propagation algorithms, called Logic-based Truth Maintenance System with Watched-literals and Incremental Truth Maintenance System with Watched-literals. These algorithms combine the strengths of the Logic-based and Incremental Truth Maintenance Systems designed for generic problem solvers with a state-of-the-art satisfiability solver data structure called watched literals. Emperical results show that the use of the watched-literals data structure significantly decreases workload of the LTMS and the ITMS without adversely affecting the incremental performance of these truth maintenance systems.

Thesis Supervisor: Brian C. Williams
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgements

I dedicate this work to my parents. Mom, you are my guiding light and my source of strength. I cannot ask for a better role model, and I know we will continue to be the closest of friends for many years to come. Daddy, thank you for all your sacrifices so that I may have the best life possible. And thank you for spoiling me with all your delicious cooking. I would not be who I am today without the two of you. I love you both.

I would also like to express my deepest thanks to my advisor Brian Williams for his guidance and support through all my ups and downs, Paul Robertson for his patience and understanding especially during the final stretches of the writing processing, Paul Elliot and Seung Chung for tirelessly answering all of my questions and helping me through the worst of debugging, and Lars Blackmore for his conversation, his humor, and for lending an ear whenever I needed someone to talk to. I truly appreciate what all of you have done for me. This thesis would not be possible without you.

Table of Contents

CHAPTER 1 INTRODUCTION 12

CHAPTER 2 THE BOOLEAN SATISFIABILITY PROBLEM AND SOLUTION... 16

2.1 SAT Problem 16

2.2 SAT Solver 17
2.2.1 Preprocessing 18
2.2.2 Decision 19
2.2.3 Deduction 21
2.2.4 Conflict Analysis 22
2.2.5 Backtracking 23

CHAPTER 3 UNIT PROPAGATION ALGORITHMS 27

3.1 Data Structures for Unit Propagation 28
3.1.1 Counter-based Approach 29
3.1.2 Head/Tail Lists 31
3.1.3 Watched-literals 33

3.2 Retracting Assignments made by Unit Propagation 37
3.2.1 Stack-based Backtracking 38
3.2.2 Logic-based Truth Maintenance System 40
3.2.2.1 Well-founded Support 40
3.2.2.2 Resupport 41
3.2.2.3 Incremental Unit Propagation with Conservative Resupport 43

3.3 Incremental Truth Maintenance System 45
3.3.1 Propagation Numbering 47
3.3.2 Conflict Repair 48
3.3.3 Aggressive Resupport 51

3.4 Root Antecedent ITMS 53
3.4.1 Root Antecedents 56
3.4.2 Conflict Repair 56
3.4.3 Aggressive Resupport 59

3.5 Summary 60

CHAPTER 4 TRUTH MAINTENANCE WITH WATCHED LITERALS 64

4.1 LTMS with watched-literals 65
4.1.1 LTMS-WL for Chronological Backtracking 67
4.1.2 LTMS-WL for Preprocessing 69

4.2 ITMS with watched literals 71
4.2.1 Conflict Repair 71
4.2.2 Aggressive Resupport 76

4.3 Summary 78

CHAPTER 5 SAT SOLVERS WITH INCREMENTAL UNIT PROPAGATION 81

5.1 ISAT 81
5.1.1 Preprocessing 82
5.1.2 Decision and Conflict Analysis 82
5.1.3 Deduction and Backtracking 83

5.2 zCHAFF 83
5.2.1 Preprocessing 83
5.2.2 Decision 84
5.2.3 Conflict Analysis 84
5.2.4 Deduction and Backtracking 85

CHAPTER 6 RESULTS AND ANALYSIS 88

6.1 Evaluation Setup 88

6.2 ISAT Performance Results 90

6.3 zCHAFF Performance Results 93

CHAPTER 7 CONCLUSION AND FUTURE WORK 97

List of Figures

Figure 1: DPLL Pseudo-code 18
Figure 2: Simple SAT Example – Preprocessing 19
Figure 3: Simple SAT Example – Decision 20
Figure 4: Simple SAT Example – Deduction 21
Figure 5: Simple SAT Example – Conflict Analysis 22
Figure 6: Simple SAT Example – Bactracking 23
Figure 7: Simple SAT Example – Decision 2 24
Figure 8: Simple SAT Example – Deduction 2 25
Figure 9: Unit Propagation Pseudo-code 28
Figure 10: Counter-based Approach Pseudo-code 30
Figure 11: Head/Tail Lists Pseudo-code 32
Figure 12: Watched Literals Pseudo-code 35
Figure 13: Stack-based Backtracking Unit Propagation Pseudo-code 38
Figure 14: Stack-based Backtracking Pseudo-code 39
Figure 15: Loop Support and Conservative Resupport Example 42
Figure 16: LTMS Unit Propagation Pseudo-code 43
Figure 17: LTMS Unassign Pseudo-code 44
Figure 18: Conservative vs Aggressive Resupport Example 46
Figure 19: Mutual Inconsistency Example 48
Figure 20: ITMS Propagation Pseudo-code 49
Figure 21: ITMS Conflict Repair Pseudo-code 50
Figure 22: ITMS Unassign Pseudo-code 52
Figure 23: ITMS Aggressive Resupport Pseudo-code 52
Figure 24: Propagation Numbering vs. Root Antecedent example 53
Figure 25: RA-ITMS Propagation Psedo-code 58
Figure 26: RA-ITMS Conflict Repair Pseudo-code 58
Figure 27: RA-ITMS Aggressive Resupport Pseudo-code 59
Figure 28: LTMS-WL Unit Propagation Pseudo-code 66
Figure 29: LTMS-WL Unassign Pseudo-code for Chronological Backtracking 68
Figure 30: LTMS-WL Unassign Pseudo-code for Preprocessing 70
Figure 31: ITMS-WL Propagation Pseudo-code 74
Figure 32: ITMS-WL Conflict Repair Pseudo-code 75
Figure 33: ITMS-WL Unassign Pseudo-code 77
Figure 34: ITMS-WL Aggressive Resupport Pseudo-code 78
Figure 35: ISAT Results Summary – Propositional Assignments 91
Figure 36: ISAT Results Summary – Clauses Visited 92
Figure 37: zCHAFF Results Summary – Propositional Assignments 94
Figure 38: zCHAFF Results Summary – Clauses Visited 95

List of Tables

Table 1: PCCA Model Properties 89
Table 2: ISAT Data 90
Table 3: zCHAFF Preprocessing Data 93

Chapter 1

Introduction

Boolean satisfiability (SAT) is the problem of determining whether there exists a satisfying assignment of variables for a propositional theory. It is a famous NP-complete problem that has been widely studied due to its theoretical significance and practical applicability. From a theoretical standpoint, SAT is viewed as the cannon NP-complete problem used to determine the sameness or difference between deterministic and nondeterministic polynomial time classes [1]. While no polynomial time SAT algorithm has been or may ever be constructed, extensive research within the SAT community produced an assortment of SAT algorithms capable of solving many interesting, real word instances.

SAT problems can be found in a variety of fields [8] including diagnosis [18, 22], planning [13], circuit testing [25], and verification [24, 12]. In many of these applications, an upper level program reduces its problem into a series of SAT theories and uses a SAT problem solver to determine their satisfiability. For example, to perform model-based diagnosis, the conflict-directed A* algorithm solves an optimal constraint satisfaction problem by testing a sequence of candidate diagnoses in decreasing order of likelihood [26]. This process is formulated as a series of tests on SAT theories denoting each candidate solution. These theories have much in common and only differ in the specific candidate solution assignments. Therefore, for these types of problems, the SAT solver must not only be able to efficiently solve a SAT problem but also efficiently perform incremental changes between similar problems.

Unit propagation is a deduction mechanism commonly used within SAT solvers to reduce computation time by pruning search spaces. It is arguably the most important component of a SAT solver that consumes over 90% of the solver's run time in most instances [20]. When dealing with a large amount of SAT theories with only small variations between them, a SAT solver can waste a considerable amount of computation time by throwing away previous results and starting unit propagation from scratch. Thus, it is desirable to utilize incremental unit propagation algorithms to increase the efficiency of a SAT solver.

A family of algorithms called truth maintenance system (TMS) can be used for this purpose [4, 19]. A TMS avoids throwing away useful results and wasting effort rediscovering the same conclusions between varying problems by maintaining justifications to variable assignments. When a change is made to the problem, the TMS algorithm uses these justifications to adjust only those variables affected by the change while leaving the rest in place.

The logic-based truth maintenance system (LTMS) is a standard TMS algorithm traditionally applied to Boolean formulas [6]. The incremental truth maintenance system (ITMS), on the other hand, increases the efficiency of the LTMS by taking a more aggressive approach during theory alterations [22]. Both algorithms have the advantage over non-incremental schemes. However, since TMS was originally designed for generic problem solvers, neither the LTMS nor the ITMS have fully exploited SAT specific properties in their design.

This thesis introduces two new algorithms called LTMS with watch-literals (LTMS-WL) and ITMS with watched-literals (ITMS-WL). These algorithms incorporate into the LTMS and the ITMS a state-of-the-art SAT data structure, called watched-literals [20]. The combined algorithms retain TMS's ability to minimize unnecessary variable unassignments during incremental updates to a theory. At the same time, the added watched-literals scheme reduces the workload of the combined algorithms by decreasing the number of clauses and variables visited during unit propagation. Empirical results show that the LTMS-WL and ITMS-WL achieve significant performance gains over an LTMS and an ITMS without watched-literals. However, when compared with a non-incremental unit propagation algorithm using watched-literals, the LTMS-WL and ITMS-WL encounters a performance tradeoff where decreasing the number of unnecessary assignments changes increases the overall workload.

For the remainder of this thesis, first Chapter 2 introduces the SAT problem, the upper level components of a SAT solver, and the basic concept behind unit propagation. The functionality of a simple SAT solver is also demonstrated through an example. Chapter 3 reviews and compares existing unit propagation data structures and unit propagation algorithms and explains why the LTMS and the ITMS along with the watched-literals data structure are chosen as the building blocks of the new algorithms. Chapter 4 details the LTMS-WL and ITMS-WL, challenges in constructing these algorithms, and their potential gains. Chapter 5 describes two SAT solvers, ISAT and zCHAFF, used for the empirical evaluation of LTMS-WL and ITMS-WL. Chapter 6 presents the testing results and analyzes the performance of LTMS-WL and ITMS-WL. And finally, Chapter 7 concludes the thesis with a discussion on potential future areas of research.

Chapter 2

The Boolean Satisfiability Problem and Solution

This chapter defines the Boolean satisfiability problem, components of a SAT Solver, and some common terminology used throughout the rest of this thesis.

2.1 SAT Problem

A propositional satisfiability problem is specified as a set of clauses in conjunctive normal form (CNF). A variable, also called a *proposition*, has Boolean domain and can be assigned values TRUE or FALSE. Given a set of propositions $P_1 \ldots P_n$, a *literal* is an instance of P_i, called a *positive literal*, or $\neg P_i$, called a *negative literal*, for $1 \le i \le n$. A *clause*, C, is a disjunction of one or more literals, $C_i = (L_{i1} \vee L_{i2} \ldots L_{ir})$, from the set of all literals $L_1 \ldots L_k$; and a SAT *theory* T is defined as $C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where m is the number of clauses. Each literal only appears once in T, but a proposition may appear multiple times. For example, $T_1 = C_1 \wedge C_2 \wedge C_3 = (L_1) \wedge (L_2 \vee L_3 \vee L_4) \wedge (L_5 \vee L_6) = (\neg P_1) \wedge (P_1 \vee P_2 \vee P_3) \wedge (P_3 \vee \neg P_4)$ is a SAT theory where $m = 3$, $k = 6$, and $n = 4$. L_1 and L_2 are both instances of P_1, and L_4 and L_5 are instances of P_3. We say that propositions and their literals are *associated* with each other—L_1 is associated with P_1 and vice versa. Incremental changes to a theory will be indicated by operators "+" and "–". For example, given T_1 from above and some clause C_4, $T_1 + C_4 - C_2 = C_1 \wedge C_3 \wedge C_4$. A *context switch* is the simultaneous addition and deletion of clauses the theory.

2.2 SAT Solver

A theory is *satisfiable* if there exists at least one truth assignment to its propositions such that the theory evaluates to true; such a truth assignment is called a satisfying assignment. A SAT solver searches for a satisfying assignment by extending partial assignments to full assignments. *Unit propagation* is a common deduction mechanism used to lighten the workload of SAT's search process. By using propositional logic to deduce variable assignments after each search step, unit propagation helps a SAT solver prune large areas of the search space with relatively little effort compared to brute force search and other deduction mechanisms.

The intuition behind unit propagation is that a theory evaluates to true if all of its clauses are true; this condition holds only if at least one literal in each clause is true. Therefore, if all but one literal in a clause evaluate to FALSE and the remaining literal is unassigned, then the proposition associated with the remaining literal should be assigned such that the literal evaluates to TRUE. For example, with clauses $C_1 = (L_1 \vee L_2) = (\neg P_1 \vee \neg P_2)$ and $C_2 = (L_3 \vee L_4) = (P_2 \vee \neg P_3)$ in a theory, C_1 is a *unit clause* if P_1 = TRUE and P_2 is unassigned. Unit propagation of C_1 will assign P_2 to FALSE so that L_2 evaluates to TRUE. At this point, C_2 becomes a unit clause, and unit propagation will assign P_3 to FALSE so that L_4 evaluates to TRUE After P_2 and P_3 are assigned, C_1 and C_2 becomes *unit propagated.* A clause is *satisfied* if at least one of its literals evaluates to true. However, if all literals within a clause evaluate to FALSE then the clause becomes *violated.* A set of variable assignments that leads to a violated clause is called a *conflict.*

Complete SAT solvers are generally based on the DPLL algorithm [3] and can be broken into five major components: preprocessing, decision, deduction, conflict analysis, and backtracking. Figure 1 shows the upper level pseudo-code for this algorithm, and the details of each component are presented in sections 2.2.1 to 2.2.5 below.

```
SAT-SOLVER(theory-T)
    if not preprocess()
        then return unsatisfiable
    while true
        do if not decide()
            then return satisfiable
        while not deduce()
            do if analyzeConflict()
                then backtrack()
                else return unsatisfiable
```

Figure 1: DPLL Pseudo-code

2.2.1 Preprocessing

Preprocessing—preprocess()—includes the addition and/or deletion of clauses, data initialization, and an initial propagation. If no theory is in place, a new one is created; else changes are made to the current theory. Preprocess() returns false if the initial unit propagation step leads to a conflict. Within the search process conflicts can be resolved by altering the value of one or more decision variables within the conflicting partial assignment. However, conflicts encountered prior to search cannot be resolved; therefore, the theory would be unsatisfiable.

Figure 2 shows a simple example where SAT theory T_2 is propagated during preprocessing. Changes between successive steps are highlighted in bold; and unit and violated clauses are italicized. In step 1 of this example, T_2 is initialized. Initially, all variables are unassigned, and C_1 is identified as a unit clause. In step 2, P_1 is assigned to TRUE by unit propagation of C_1, and C_2 is identified as a unit clause. Finally, in step 3, P_2 is assigned to TRUE by unit propagation of C_2, and there are no more unit clauses. At this point unit propagation terminates and preprocessing is complete. Since P_1 and P_2 are assigned during preprocessing, any complete satisfying assignment to T_2 must contain the partial assignment $\{P_1 = \text{TRUE}; P_2 = \text{TRUE}\}$.

Step 1

$$T_2 = \begin{array}{l} C_1 : (P_1) \wedge \\ C_2 : (\neg P_1 \vee P_2) \wedge \\ C_3 : (\neg P_1 \vee \neg P_3 \vee P_4) \wedge \\ C_4 : (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

$\Rightarrow$

Step 2

$$\begin{array}{l} C_1 : (\mathbf{P_1 = TRUE}) \wedge \\ C_2 : (\neg \boldsymbol{P_1} = \boldsymbol{FALSE} \vee P_2) \wedge \\ C_3 : (\neg \mathbf{P_1 = FALSE} \vee \neg P_3 \vee P_4) \wedge \\ C_4 : (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

$\Downarrow$

Step 3

$$\begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee \mathbf{P_2 = TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 \vee P_4) \wedge \\ C_4 : (\neg \mathbf{P_2 = FALSE} \vee \neg P_3 \vee \neg P_5) \wedge \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

Figure 2: Simple SAT Example – Preprocessing

Had T_2 contained an additional clause $C_6 = (\neg P_1 \vee \neg P_2)$, C_6 would become violated by the end of step 3. In such an event, the partial assignment $\{P_1 = \text{TRUE}; P_2 = \text{TRUE}\}$ would be a conflict. Since conflicts encountered during preprocessing cannot be resolved, a SAT solver would find that T_2 is unsatisfiable.

2.2.2 Decision

The search decision algorithm—decide()—chooses the next variable for the search to branch on and the value it takes; this may involve any number of heuristics including random selection [15, 10], conflict analysis of previous results [21], and dynamic updates of current states [15, 10]. A proposition, P, assigned by the decision algorithm, and not through unit propagation, is called a *decision variable*. If the decision algorithm is called when no unassigned variables remain, decide() returns false. This only happens when all variables are assigned, and there are no violated clauses; thus the theory is satisfiable.

Figure 3 shows the decision step following preprocessing of the simple SAT example introduced in Figure 2. This example uses a very simple decision algorithm that works in the following ways:

1. Decision variables are always assigned to TRUE before FALSE.
2. When called at the end of a deduction step where no clauses are violated, decide() selects an unassigned variable P and assigns P to TRUE.
3. When called after conflict analysis and backtracking, decide() takes the decision variable P selected by analyzeConflict() and assigns P to FALSE. The conflict analysis algorithm for this example is presented in section 2.2.4.

Step 3

$$\mathbf{T_2} = \begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 \vee P_4) \wedge \\ C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 \vee \neg P_5) \wedge \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

$$\Downarrow$$

Step 4

$$\begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \mathit{FALSE} \vee \boldsymbol{\neg P_3 = \mathit{FALSE}} \vee P_4) \wedge \\ C_4 : (\neg P_2 = \mathit{FALSE} \vee \boldsymbol{\neg P_3 = \mathit{FALSE}} \vee \neg P_5) \wedge \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

Figure 3: Simple SAT Example – Decision

Step 3 in Figure 3 is the same as Figure 2 where preprocessing is complete and no clauses are violated. In Step 4, the algorithm selects an unassigned proposition—in this case P_3—and assigns P_3 to TRUE. C_3 and C_4 are both identified as unit clauses.

2.2.3 Deduction

The deduction algorithm—deduce()—is invoked after the decision step. Unit propagation is the most commonly used deduction mechanism; but other techniques such as the pure literal rule exist [31]. If deduction terminates without encountering a violated clause, the algorithm proceeds for another decision step. Deduce() returns false if it encounters a violated clause, at which point the conflict analysis and resolution algorithm—analyzeConflict()—is invoked. Figure 4 continues the simple SAT example with a deduction step. For this example, deduction consists of only unit propagation.

Step 4

$$\mathbf{T_2} = \begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ \mathit{C_3 : (\neg P_1 = FALSE \vee \neg P_3 = FALSE \vee P_4) \wedge} \\ \mathit{C_4 : (\neg P_2 = FALSE \vee \neg P_3 = FALSE \vee \neg P_5) \wedge} \\ C_5 : (\neg P_4 \vee P_5) \end{array}$$

$$\Downarrow$$

Step 5

$$\begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{FALSE} \vee \mathbf{P_4 = TRUE}) \wedge \\ \mathit{C_4 : (\neg P_2 = FALSE \vee \neg P_3 = FALSE \vee \neg P_5) \wedge} \\ \mathit{C_5 : (\boldsymbol{\neg P_4 = FALSE} \vee P_5)} \end{array}$$

$$\Downarrow$$

Step 6

$$\begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{FALSE} \vee P_4 = \text{TRUE}) \wedge \\ C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \text{FALSE} \vee \mathbf{\neg P_5 = TRUE}) \wedge \\ \mathit{C_5 : (\neg P_4 = FALSE \vee \boldsymbol{P_5 = FALSE})} \end{array}$$

Figure 4: Simple SAT Example – Deduction

By the end of step 4, C_3 and C_4 had both been identified as unit clauses. In step 5, P_4 is assigned to TRUE by unit propagation of C_3, and C_5 is identified as a unit clause. In step 6,

P_5 is assigned to TRUE by unit propagation, and C_5 becomes violated. At this point, deduction terminates and the conflict analysis algorithm is called.

2.2.4 Conflict Analysis

As stated earlier, a conflict can be resolved by altering the value of one or more search variables within the conflicting partial assignment. In the simplest case, analyzeConflict() looks for the latest decision variable whose entire domain (TRUE or FALSE) has not been searched over and passes this information on to the backtracking and decision algorithms. A more advanced conflict analysis algorithm will identify and prune search space that generates the conflict so that the same conflict will not be encountered again [16, 30]. If the entire domain of all search variables within the conflicting partial assignment has been searched over, then the conflict cannot be resolved. In these situations, analyzeConflict() returns false and the theory is unsatisfiable.

Step 6

$$
\mathbf{T_2} = \begin{array}{l}
C_1 : (P_1 = \text{TRUE}) \wedge \\
C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\
C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{FALSE} \vee P_4 = \text{TRUE}) \wedge \\
C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \text{FALSE} \vee \neg P_5 = \text{TRUE}) \wedge \\
\mathit{C_5 : (\neg P_4 = FALSE \vee P_5 = FALSE)}
\end{array}
$$

$$\Downarrow$$

Step 7

$$
\begin{array}{l}
C_1 : (P_1 = \text{TRUE}) \wedge \\
C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\
C_3 : (\neg P_1 = \text{FALSE} \vee \mathit{\neg P_3 = FALSE} \vee P_4 = \text{TRUE}) \wedge \\
C_4 : (\neg P_2 = \text{FALSE} \vee \mathit{\neg P_3 = FALSE} \vee \neg P_5 = \text{TRUE}) \wedge \\
\mathit{C_5 : (\neg P_4 = FALSE \vee P_5 = FALSE)}
\end{array}
$$

Figure 5: Simple SAT Example – Conflict Analysis

Figure 5 extends the simple SAT example with a conflict analysis step. This example uses a simple conflict analysis algorithm that looks for the most recently assigned decision variable of value TRUE. Recall the decision algorithm used for this example (see section

2.2.2) always assigns a decision variable to TRUE before FALSE. Therefore, decision variables of value TRUE have not been assigned FALSE, while decision variables of value FALSE have had their entire domain searched over. This conflict analysis algorithm simply identifies a decision variable assigned to TRUE but does not alter the assignment of any variables.

C_5 was identified as a violated clause in step 6. The conflict responsible for violating C_5 is $\{P_1$ = TRUE; P_2 = TRUE; P_3 = TRUE; P_4 = TRUE; P_5 = FALSE$\}$. In step 7, P_3 is identified as latest decision variable assigned TRUE, and conflict analysis terminates. Since P_3 is also the only decision variable in this case, had it's value been FALSE, the conflict would not be resolvable, and T_2 would be unsatisfiable.

2.2.5 Backtracking

Once analyzeConflict() identifies a decision variable, P_D, that can resolve the conflict, the backtracking algorithm—backtrack()—unassigns P_D and all propositions assigned after P_D. Backtracking, as we define it, has no control over which point in the search tree SAT returns to during conflict analysis—that task is left up to analyzeConflict().

Step 7

$$
\mathbf{T_2} = \begin{array}{l}
C_1 : (P_1 = \text{TRUE}) \wedge \\
C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\
C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \textit{FALSE} \vee P_4 = \text{TRUE}) \wedge \\
C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \textit{FALSE} \vee \neg P_5 = \text{TRUE}) \wedge \\
C_5 : (\neg P_4 = \text{FALSE} \vee P_5 = \text{FALSE})
\end{array}
$$

$\Downarrow$

Step 8

$$
\begin{array}{l}
C_1 : (P_1 = \text{TRUE}) \wedge \\
C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\
C_3 : (\neg P_1 = \text{FALSE} \vee \mathbf{\neg P_3} \vee \mathbf{P_4}) \wedge \\
C_4 : (\neg P_2 = \text{FALSE} \vee \mathbf{\neg P_3} \vee \mathbf{\neg P_5}) \wedge \\
C_5 : \mathbf{(\neg P_4 \vee P_5)}
\end{array}
$$

Figure 6: Simple SAT Example – Bactracking

Step 8

$\mathbf{T_2} =$

$C_1 : (P_1 = \text{TRUE}) \wedge$
$C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge$
$C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 \vee P_4) \wedge$
$C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 \vee \neg P_5) \wedge$
$C_5 : (\neg P_4 \vee P_5)$

$\Downarrow$

Step 9

$C_1 : (P_1 = \text{TRUE}) \wedge$
$C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge$
$C_3 : (\neg P_1 = \text{FALSE} \vee \mathbf{\neg P_3 = TRUE} \vee P_4) \wedge$
$C_4 : (\neg P_2 = \text{FALSE} \vee \mathbf{\neg P_3 = TRUE} \vee \neg P_5) \wedge$
$C_5 : (\neg P_4 \vee P_5)$

$\Downarrow$

Step 10

$C_1 : (P_1 = \text{TRUE}) \wedge$
$C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge$
$C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee \mathbf{P_4 = TRUE}) \wedge$
$C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee \neg P_5) \wedge$
$C_5 : (\boldsymbol{\neg P_4 = FALSE} \vee P_5)$

Figure 7: Simple SAT Example – Decision 2

Figure 6 demonstrates backtracking in the simple SAT example. In step 8, P_3, P_4, and P_5 are unassigned, and backtracking terminates.

In Figure 7, the simple SAT example returns to the decision algorithm. Recall that when called after conflict analysis and backtracking, the decision algorithm takes the decision variable selected by analyzeConflict()—in this case P_3—and assigns P_3 to FALSE. In step 9, no unit clauses are generated after P_3 is assigned FALSE; therefore, the decision algorithm selects another unassigned proposition, P_4. In step 10, P_4 is assigned to TRUE, and C_5 becomes a unit clause.

Figure 8 presents the final deduction step in the simple SAT example:

Step 10

$$T_2 = \begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee P_4 = \text{TRUE}) \wedge \\ C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee \neg P_5) \wedge \\ \mathit{C_5 : (\neg P_4 = FALSE \vee P_5)} \end{array}$$

$$\Downarrow$$

Step 11

$$\begin{array}{l} C_1 : (P_1 = \text{TRUE}) \wedge \\ C_2 : (\neg P_1 = \text{FALSE} \vee P_2 = \text{TRUE}) \wedge \\ C_3 : (\neg P_1 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee P_4 = \text{TRUE}) \wedge \\ C_4 : (\neg P_2 = \text{FALSE} \vee \neg P_3 = \text{TRUE} \vee \mathbf{\neg P_5 = FALSE}) \wedge \\ C_5 : (\neg P_4 = \text{FALSE} \vee \mathbf{P_5 = TRUE}) \end{array}$$

Figure 8: Simple SAT Example – Deduction 2

Step 11 assigns P_5 to TRUE. At this point, all variables are assigned, and no conflicts are found. Therefore, the SAT theory T_2 is satisfiable with satisfying assignment {P_1 = TRUE; P_2 = TRUE; P_3 = FALSE; P_4 = TRUE; P_5 = TRUE}, though other satisfying assignments may exist.

Chapter 3

Unit Propagation Algorithms

Vast amounts of research efforts have been committed to the study of the Boolean satisfiability (SAT) problem and to the optimization of its solution approach. Chapter 2 provided an overview of the SAT problem and the typical structure of a SAT solver. A comprehensive study of all aspects of the SAT solution is beyond the scope of our work. Instead, for the remainder of this thesis, we will focus on the unit propagation algorithm used by a SAT solver during preprocessing, deduction, and backtracking.

Unit propagation is an effective deduction mechanism that has been widely adapted by SAT solvers. Although the basic concept behind unit propagation is simple, there are many variations to its implementation that can lead to vastly different performance results. This chapter reviews some existing unit propagation algorithms and their relative strengths and weaknesses. Section 3.1 introduces three different data structures used for unit propagation: counter-based approach [14], head/tail lists [28, 29], and watched-literals [20], [31]. A difference in data structures affects the number of clauses a unit propagation algorithm visits after each variable assignment, and how the algorithm determines whether or not a clause is unit. Section 3.2 presents two different unit propagation backtracking algorithms used to retract variable assignments made during unit propagation: stacked-based backtracking [20] and logic-based truth maintenance system (LTMS) [6, 27]. The former is a non-incremental algorithm often used with backtrack search. The later is an incremental algorithm adapted from traditional truth maintenance schemes to handle propositional clauses used in SAT theories. Sections 3.3 and 3.4 present two incremental truth maintenance systems, called ITMS [22] and Root Antecedent ITMS [27] respectively. An ITMS takes a more aggressive incremental approach than the LTMS in order to achieve higher efficiency by reducing the number of unnecessary variable unassignments. Finally, section 3.5 summarizes these algorithms and why LTMS, ITMS, and watched-literals are used as the building blocks for the new algorithms presented in Chapter 4.

3.1 Data Structures for Unit Propagation

The logic behind unit propagation was described in the previous section. Figure 9 below presents the pseudo-code for unit propagation. After a variable assignment, the algorithm checks each clause containing a literal associated with the variable and propagates if a clause is unit. Recall that a unit clause contains one unassigned literal, while all remaining literals evaluate to FALSE. Therefore, if proposition P in Figure 9 is assigned TRUE then only clauses with negative literals associated with P may become unit (or violated)—the opposite holds for P = FALSE. Unit propagation terminates when there are no more unit clauses or when a clause is violated.

```
propagate(P)
1       if P = TRUE
2               then CP ← list of clauses with negative literals associated with P
3               else CP ← list of clauses with positive literals associated with P
4       for i ← 1 to length(CP)
5               do if CP[i] is a unit clause
6                       then P1 ← proposition in CP[i] with value UNKNOWN
7                            if literal associated with P1 is POSITIVE
8                               then P1 ← TRUE
9                               else P1 ← FALSE
10                           if propagate(P1) = false
11                              then return false
12                  if CP[i] is violated
13                      then return false
14      return true
```
Figure 9: Unit Propagation Pseudo-code

The forward, propagation phase of unit propagation algorithms differ primarily in the number of clauses searched per propositional assignment (Figure 9 lines 1 to 3) and how a clause is identified as unit or violated (Figure 9 line 5). The number of clauses searched, in particular, can greatly affect the performance of the algorithm. An adjacency list data structure [14] such as the counter-based approach looks at all clauses containing literals associated with a newly assigned proposition. Lazy data structures such as head/tail lists and watched-literals, on the other hand, conserve computation time by only looking at a subset of those clauses. The details of these algorithms are presented in sections 3.1.1 through 3.1.3 below.

3.1.1 Counter-based Approach

The counter-based approach [14] is one of the earlier data methods used to identify unit and violated clauses and is a good benchmark algorithm to compare the more recent lazy data structures against. While the earliest algorithms simply recheck every literal in a clause to determine if the clause is unit, the counter-based method stores with each clause its current number of TRUE literals and its current number of FALSE literals; the number of unassigned literals in the clause can be deduced from this information.

Figure 10 contains pseudo-code for the counter-based approach. When a proposition P is assigned, all clauses with literals associated with P update their number-of-FALSE-literals, N^F, or number-of-TRUE-literals, N^T, depending on whether the associated literal is positive or negative (Figure 10 lines 8 and 22). For example, consider clauses $C_1 = (\neg P_1 \vee P_2 \vee P_3)$ and $C_2 = (\neg P_1 \vee P_2)$, where C_1 and C_2 are part of some larger theory. The total number of literals in C_1 and C_2 are $N_1 = 3$ and $N_2 = 2$ respectively. Initially, P_1, P_2, and P_3 are all unassigned; therefore, counters $N^T{}_1 = N^T{}_2 = N^F{}_1 = N^F{}_2 = 0$. Now, assume unit propagation of some other clause in the theory led to the assignment P_1 = FALSE. Since, C_1 and C_2 each contain a positive instance of P_1, which evaluates to FALSE after the assignment, $N^F{}_1$ and $N^F{}_2$ become 1. Next, if P_2 is assigned TRUE due to unit propagation of another clause in the theory, then $N^F{}_1 = N^F{}_2 = 2$ because C_1 and C_2 each contain a negative instance of P_2 which evaluates to FALSE after the assignment; the other counters $N^T{}_1$ and $N^T{}_2$ remains 0.

A clause is unit if the number-of-TRUE-literals equals 0 and the number-of-FALSE-literals is one less than the total number of literals in the clause (line 12). A clause is violated if the number-of-TRUE-literals is 0 and the number-of-FALSE-literals equals the total number of literals in the clause (line 19). For the example above, after the assignments P_1 = FALSE and P_2 = TRUE, counters for C_1 and C_2 took on the values $N^T_1 = N^T_2 = 0$ and $N^F_1 = N^F_2 = 2$. Since, $N^T_1 = 0$ and $N^F_1 = 2$ is one less than $N_1 = 3$, C_1 is a unit clause. C_2 on the other hand is violated because $N^T_2 = 0$ and $N^F_2 = N_2 = 2$.

```
propagate(P)
1       if P = TRUE
2               then Cp^T ← list of all clauses with positive literals associated with P
3                    Cp^F ← list of all clauses with negative literals associated with P
4               else Cp^T ← list of all clauses with negative literals associated with P
5                    Cp^F ← list of all clauses with positive literals associated with P
6       noConflict ← true
7       for i ← 1 to length(Cp^F)
8               do increment number-of-FALSE-literals in Cp^F[i] by 1
9                  N^T_i ← number-of-TRUE-literals in Cp^F[i]
10                 N^F_i ← number-of-FALSE-literals in Cp^F[i]
11                 N_i ← total-number-of-literals in Cp^F[i]
12                 if N^T_i = 0 and N^F_i = N_i - 1
13                      then P_1 ← proposition in Cp^F[i] with value UNKNOWN
14                              if literal associated with P_1 is POSITIVE
15                                 then P_1 ← TRUE
16                                 else P_1 ← FALSE
17                              if propagate(P_1) = false
18                                 then noConflict ← false
19                 if N^T_i = 0 and N^F_i = N_i
20                      then noConflict ← false
21      for j ← 1 to length(Cp^T)
22              do increment number-of-TRUE-literals in Cp^F[i] by 1
23      return noConflict
```
Figure 10: Counter-based Approach Pseudo-code

When a proposition gets unassigned due to search backtrack or clause deletion, all clauses containing associated literals must update their counters. For example, if P_1 and P_2 are unassigned, C_1 and C_2 must reset their counters so that $N^F_1 = N^F_2 = 0$. Maintaining the counter values during variable unassignment requires roughly the same work as variable assignment [31].

The specific counters used for this approach may vary without affecting the performance of the algorithm. For example, the number of false literals can be replaced by the number of unknown literals. In this case, a clause would be unit if the number of true literals equals zero and the number of unknown literals equals one; a clause would be violated if the number of true literals equals zero and the number of unknown literals equals zero. However, regardless of the specific counters used, this approach requires that an update be performed to every clause associated with a newly assigned or unassigned proposition.

3.1.2 Head/Tail Lists

An alternative approach for efficiently detecting unit and violated clauses is to use head/tail lists [28, 29, 31]. A key contribution of head/tail lists is the notion that a unit propagation algorithm need not search through all clauses containing a newly assigned proposition to identify all unit and violated clauses. This algorithm is not used in this thesis as a benchmark or part of the new algorithms. However, its details are presented below because it shares many common traits with the watched-literals approach in section 3.1.3.

A head/tail lists algorithm maintains pointers to two literals for each clause with two or more literals. Initially, all literals are unassigned. The head literal is the first literal in a clause, and the tail literal is the last literal. For example, given clause $C_1 = (\mathbf{P_1} \vee \neg P_2 \vee P_3 \vee \neg \mathbf{P_4})$ that is part of some larger theory, P_1 equals the head literal, and $\neg P_4$ equals the tail literal of C_1.

A clause cannot be unit or violated if it contains at least two unassigned literals. Therefore, no matter what values are assigned to propositions P_2 and P_3, C_1 can neither be unit nor violated as long as its head literal, P_1, and its tail literal, $\neg P_4$, are unassigned. For this reason, a clause will only need to be visited during unit propagation if a newly assigned proposition is associated with the head or tail literals of the clause. Furthermore, only instances where the head/tail literal evaluates to FALSE are considered, because if either P_1 or $\neg P_4$ evaluates to TRUE, then C_1 cannot be unit or violated.

Figure 11 presents the pseudo code for unit propagation with head/tail lists. When a proposition P is assigned, the algorithm only looks at the list of clauses, C_P, where each $C_P[i]$ contains a head or tail literal associated with P and that literal evaluates to FALSE (lines 1 to 3). C_P is on average shorter than the list of all literals by a ratio of 1:average-number-of-literals-per-clause.

```
propagate(P)
1     if P = TRUE
2            then C_P ← list of clauses with negative head/tail literals associated with P
3            else C_P ← list of clauses with positive head/tail literals associated with P
4     noConflict ← true
5     for i ← 1 to length(C_P)
6            do L^H ← head literal in C_P[i]
7               L^T ← tail literal in C_P[i]
8               for each L between L^H and L^T starting from L^H/L^T
9                   do if L = TRUE
10                          then break
11                     if L = UNKNOWN and L ≠ L^T/L^H
12                          then L is the new head/tail literal
13                               insert C_P[i] into L's head/tail list
14                               break
15                     if L = L^T/L^H = UNKNOWN
16                          then P_1 ← proposition associated with L
17                                 if L is POSITIVE
18                                      then P_1 ← TRUE
19                                      else P_1 ← FALSE
20                                 if propagate(P_1) = false
21                                      then noConflict ← false
22                     if L = L^T/L^H = FALSE
23                          then noConflict ← false
24    return noConflict
```

Figure 11: Head/Tail Lists Pseudo-code

For head literal L^H in clause $C_P[i]$ that is associated with P and evaluates to FALSE, the algorithm searches for the first unknown literal in $C_P[i]$ to be the new head literal (lines 11-14). However, if the search first encounters a TRUE literal, then $C_P[i]$ is satisfied and can no longer be unit or violated, so a new head literal is not needed (lines 9-10). For example, given C_1 from above, assume that at some point P_2 had been assigned to TRUE so that $C_1 = (\mathbf{P_1} \vee \neg P_2 = \text{FALSE} \vee P_3 \vee \neg\mathbf{P_4})$. (Since $\neg P_2$ is not a head/tail literal of C_1, C_1 was not visited during P_2's assignment.) If P_1 is assigned FALSE, the head/tail lists algorithm will search for another head literal starting from literal $\neg P_2$. Since $\neg P_2$ = FALSE, the search moves on to P_3. P_3 = UNKNOWN; therefore, P_3 becomes the new head literal of C_1: $C_1 = (P_1\text{=FALSE} \vee \neg P_2 = \text{FALSE} \vee \mathbf{P_3} \vee \mathbf{\neg P_4})$. Had $\neg P_2$ = TRUE, there would be no need to find P_3.

Under this scheme $C_P[i]$ is unit if the head literal equals the tail literal and the associated proposition is unassigned (lines 15-21). $C_P[i]$ is violated if the head literal equals the tail literal and the associated proposition evaluates to FALSE (lines 22-23). For example, if unit propagation of some other clause led to the assignment P_4 = TRUE, then $C_1 = (P_1\text{=FALSE} \vee \neg P_2 = \text{FALSE} \vee \mathbf{P_3} \vee \mathbf{\neg P_4 = FALSE})$. Since $\neg P_4$ is the tail literal in C_1, head/tail lists searches for another unassigned literal. In this case, the first literal encountered is the head literal P_3. P_3 is unassigned, and therefore, C_1 is a unit clause. Had unit propagation of another clause assigned P_3 to FALSE, then C_1 would be violated.

There is no need to search through literals P_1 and $\neg P_2$ because they are located before the head literal. Since the head and tail literals are the first and last unassigned literals in a clause and are not changed when assigned TRUE, all other literals not in-between the head and tail must evaluate to FALSE.

Head/tail pointer must also be updated during variable unassignment. For example, if P_1 through P_5 are all unassigned, $C_1 = (P_1 \vee \neg P_2 \vee \mathbf{P_3} \vee \mathbf{\neg P_4})$. However, since P_1 is unassigned, P_3 is no longer first unassigned literal in the clause. Therefore, the head pointer must be reverted back to P_1 so that $C_1 = (\mathbf{P_1} \vee \neg P_2 \vee P_3 \vee \mathbf{\neg P_4})$. The workload for updating head/tail lists during variable unassignment is roughly the same as variable assignment [31].

3.1.3 Watched-literals

The watched-literals scheme is one of the most recently developed SAT data structures [20, 31]. Like the head/tail lists, watched-literals is a lazy data structure that allows unit propagation to only search through a subset of the clauses containing a newly assigned proposition. And like the head/tail lists, the watched-literals scheme places special emphasis on two literals per clause, called the watched literals. However, in this case the watched literals can be any two non-false literals in the clause. Also, when a watched literal is assigned FALSE, the algorithm may select any non-FALSE literal to replace the watch. Due to this flexibility in placement, the watched literals need not be updated after variable unassignments, an advantage over the head/tail lists. For example, recall C_1 = (P_1=FALSE $\vee$ $\neg P_2$ = FALSE $\vee$ $\mathbf{P_3}$ $\vee$ $\mathbf{\neg P_4}$ **= FALSE**) from section 3.1.2 where the head and tail literals are in bold. When propositions P_1 through P_5 were unassigned, the head literal must be reverted back to P_1. However, with the watched-literals scheme, P_3 and $\neg P_4$ can remain as the watched literals even if all propositions are unassigned so that $C_1 = (P_1 \vee \neg P_2 \vee \mathbf{P_3} \vee \mathbf{\neg P_4})$.

Figure 12 presents the pseudo-code for the watched-literals algorithm. When a watched literal is assigned FALSE, the algorithm attempts to shift the watch to any non-false, unwatched literal in the clause if one exists by searching through all literals in the clause. The watched literal remains FALSE only if no such unwatched literal can be found (line 8). For example, $C_1 = (P_1 \vee \neg P_2 \vee \mathbf{P_3} \vee \mathbf{\neg P_4})$, where all propositions are unassigned and literals P_3 and $\neg P_4$ are watched. If unit propagation of some other clause leads to the assignment P_4 = TRUE, then watched literal $\neg P_4$ becomes FALSE. A watched-literals scheme searches for any unwatched, non-false literal—in this case P_1—to replace the watch, so that C_1 = ($\mathbf{P_1}$ $\vee$ $\neg P_2$ $\vee$ $\mathbf{P_3}$ $\vee$ $\neg P_4$ = FALSE). Next, assume P_2 is assigned to TRUE. Since, $\neg P_2$ is not watched in C_1, C_1 is not visited after P_2's assignment. Finally, assume P_3 is assigned to FALSE. Again, the algorithm searches for any unwatched, non-false literal in C_1. Had P_2 been assigned FALSE so that $\neg P_2$ = TRUE, $\neg P_2$ would have been selected as the new watch. However, since all unwatched literals in C_1 are FALSE, P_3 remains watched so that C_1 = ($\mathbf{P_1}$ $\vee$ $\neg P_2$ = FALSE $\vee$ $\mathbf{P_3}$ **= FALSE** $\vee$ $\neg P_4$ = FALSE). Locating a new watched literal generally takes more work then locating a new head/tail literal because head/tail literals need not consider literals not between the head and tail while watched literals must consider all literals in a clause each time a watch needs to be replaced.

```
propagate(P)
1     if P = TRUE
2           then C_P ← list of clauses with negative watched literals associated with P
3           else C_P ← list of clauses with positive watched literals associated with P
4     noConflict ← true
5     for i ← 1 to length(C_P)
6           do W^1 ← watched literal in C_P[i] associated with P
7              W^2 ← other wathed literal in C_P[i]
8              replace W^1 with non-FALSE, unwatched literal if possible
9              if W^1 cannot be replaced and W^2 = UNKNOWN
10                 then P_1 ← proposition associated with W^2
11                      if W^2 is POSITIVE
12                         then P_1 ← TRUE
13                         else P_1 ← FALSE
14                      if propagate(P_1) = false
15                         then noConflict ← false
16             if W^1 cannot be replaced and W^2 = FALSE
17                 then noConflict ← false
18    return noConflict
```

Figure 12: Watched Literals Pseudo-code

Under the watched literals scheme, a clause is unit if one watched literal is FALSE and the other unassigned (line 9). A clause is a violated if both watched literals are FALSE (line 16). Therefore, after P_3's assignment, $C_1 = (\mathbf{P_1} \vee \neg P_2 = \text{FALSE} \vee \mathbf{P_3 = FALSE} \vee \neg P_4 = \text{FALSE})$ became a unit clause. If unit propagation of some other clause leads to the assignment $P_1 = \text{FALSE}$, then C_1 would be violated.

As stated earlier, watched-literals has the advantage over head/tail lists in that the watched literals need not be updated during backtracking. However, this is dependent on the condition that the last literals assigned during propagation must be the first unassigned during backtracking. If this condition is violated then there might be clauses where the watched literals are FALSE while some non-watched literals are unassigned. For example, $C_1 = (\mathbf{P_1 = FALSE} \vee \neg P_2 = \text{FALSE} \vee \mathbf{P_3 = FALSE} \vee \neg P_4 = \text{FALSE})$. Recall that the variables were assigned in the following order: $P_4 = \text{TRUE}$, $P_2 = \text{TRUE}$, $P_3 = \text{FALSE}$, and $P_1 = \text{FALSE}$; so P_1 is the proposition last assigned. If we unassign the propositions P_1, P_3, and P_4 in order, then C_1 becomes ($\mathbf{P_1} \vee \neg P_2$ = FALSE $\vee$ $\mathbf{P_3} \vee \neg P_4$); in this case, the conditions for watched literals are met because the watched literals P_1 and P_3 are both unassigned. However, if P_3 and P_4 are unassigned without unassigning P_1 then C_1 becomes ($\mathbf{P_1}$ **= FALSE** $\vee \neg P_2$ = FALSE $\vee$ $\mathbf{P_3} \vee \neg P_4$); in this case, the watched-literals rules are violated because P_1 = FALSE is watched while there exists unassigned literal $\neg P_4$ in the same clause.

Backtracking in reverse order eliminates the need to update watched literals during backtracking for the following reasons:

1. If a watched literal was originally TRUE or unassigned, then it cannot become FALSE during backtracking, and thus need not be updated;
2. If a watched literal was originally FALSE, then it must be assigned AFTER all unwatched literals in the same clause, because a FALSE literal can remain watched only if all unwatched literals in the clause are already FALSE. Therefore, as long as backtracking unassigns variables in reverse order, FALSE watched literals will become unassigned before the unwatched literals, and therefore, need not be updated.

In summary, a watched-literals scheme has the advantage over a counter-based based approach because the former only looks at a subset of clauses associated with a newly assigned proposition while the latter must update all clauses containing that proposition. It also has the edge over head/tail lists because, unlike head/tail literals, the watched literals' pointers do not need to be update during backtracking. One disadvantage of the watched-literals is that updating watched literals pointers requires more work than updating head/tail literals because watched literals may be replaced with any unwatched literal in a clause while head/tail literals could only be replaced with literals in-between the head and tail literals. However, empirical results show that despite this drawback, a watched-literals scheme still out performs both the head/tail lists and the counter-based approach across a variety of SAT problems [14].

3.2 Retracting Assignments made by Unit Propagation

Within a SAT solver, assignments previously asserted by a unit propagation algorithm may latter be retracted due to one of two situations:

1. A clause that *supports* a proposition is deleted from the theory. A clause C supports a proposition P if the value of P resulted from unit propagation of C. If $C_1 = (P_1)$ supports P_1 = TRUE, and C_1 is deleted from the theory, then P_1 must be unassigned. As a result, all propositions *dependent* on P_1 must also be unassigned. We say that a proposition P' is dependent on P if P' is assigned through unit propagation of a clause containing P; propositions dependent on P' are also dependent on P. For example, if P_1 = TRUE and $C_2 = (\neg P_1 = \text{FALSE} \vee P_2)$, then unit propagation of C_2 will assign P_2 to TRUE. Hence, the assignment P_2 = TRUE is dependent on P_1 = TRUE. When P_1 is unassigned, $C_1 = (\neg P_1 = \text{UNKNOWN} \vee P_2 = \text{TRUE})$ will no longer be able to support P_2, and P_2 must be unassigned. And when P_2 is unassigned, all variables dependent on P_2 must be unassigned as well.
2. A decision variable P_D is unassigned. (Recall, a decision variable is a variable whose truth-value was decided explicitly by the search and not through unit propagation.) When this happens, all propositions dependent on P_D must be unassigned.

Section 3.2 provides two algorithms used to retract unit propagation: stack-based backtracking and LTMS. The former is a non-incremental algorithm that will be used as a baseline benchmark for our new algorithms; and the latter is an incremental algorithm used as a component of the new LTMS with watched-literals algorithm introduced in section 4.1.

3.2.1 Stack-based Backtracking

The stacked-based backtracking algorithm is used by SAT solvers to retract unit propagation assignments during tree search when the search algorithm removes assignments to decision variables [20]. The key to stack-based backtracking is the level within the search tree that we call the decision level. All propositions assigned during preprocessing belong to decision level zero; propositions assigned during and after the first search decision, including the decision variable, belong to decision level one and so on. All assignments made after a new decision P_D and before the next decision P_D' are the result of unit propagation of P_D. Therefore, if P_D is the decision variable at decision level DL, then all other assignments made at DL must be dependent on P_D and should be unassigned when the value of P_D changes.

```
propagate(P)
1       if P = TRUE
2               then C_P ← list of clauses with negative literals associated with P
3               else C_P ← list of clauses with positive literals associated with P
4       for i ← 1 to length(C_P)
5               do if C_P[i] is a unit clause
6                       then P_1 ← proposition in C_P[i] with value UNKNOWN
7                            if literal associated with P_1 is POSITIVE
8                               then P_1 ← TRUE
9                               else P_1 ← FALSE
10                           DL ←current decision level
11                           push(P_1, assignmentStackList[DL])
12                           if propagate(P_1) = false
13                              then return false
14                  if C_P[i] is violated
15                      then return false
16      return true
```

Figure 13: Stack-based Backtracking Unit Propagation Pseudo-code

Figure 13 provides the unit propagation pseudo-code modified to accommodate stack-based backtracking. The only difference between this algorithm and the one presented in Figure 9 are lines 10-11, where a newly assigned proposition P_1 is pushed onto the assignment stack corresponding to the decision level at which P_1 is assigned.

Figure 14 presents the pseudo-code for stack-based backtracking. After some conflict resolution algorithm decides the decision level, DL, to retract to, backtrack(DL) simply unassigns all propositions assigned during a search level greater than or equal to DL.

```
backtrack(DL)
    ▷ assignmentStackList[i] is the stack of all assignments at decision level i
    for i ← length(assignmentStackList) to DL
        do while not empty(assignmentStack[i])
            do P ←top(assignmentStack[i])
               P ← UNKNOWN
               pop(assignmentStack[i])
        remove(assignmentStackList[i], assignmentStackList)
```

Figure 14: Stack-based Backtracking Pseudo-code

This algorithm can only be used for chronological backtracking. Since there is no way to identify the relationship between propositions and supports, decision levels must be backtracked in sequence to maintain soundness. For example, clause $C_1 = (\neg P_1 \vee P_2 \vee P_3)$ is part of some larger theory. Assume P_1 is assigned TRUE by unit propagation of some other clause at DL 3. At DL 4, P_2 is assigned FALSE, and $C_1 = (\neg P_1 = \text{FALSE} \vee P_2 = \text{FALSE} \vee P_3)$ becomes a unit clause. Unit propagation of C_1 at DL 4 will assign P_3 to TRUE. Therefore, P_1 is on the assignment stack for DL 3, and P_2 and P_3 are on the assignment stack for DL 4. If the search is backtracking chronologically, unassigning variables at DL 4 will revert C_1 to $(\neg P_1 = \text{FALSE} \vee P_2 \vee P_3)$. However, if backtracking takes place out of order, and DL 3 is backtracking while DL 4 is not, then C_1 becomes $(\neg P_1 \vee P_2 = \text{FALSE} \vee P_3 = \text{TRUE})$, even though, without $P_1 = \text{TRUE}$, C_1 cannot support $P_3 = \text{TRUE}$.

As mentioned earlier, stack-based backtracking is a non-incremental algorithm. When a clause is deleted from the theory, the stack-based method has no way of isolating the dependents of this clause; therefore, the entire assignment stack must be backtracked. For example, clauses $C_2 = (\neg P_1)$, $C_3 = (P_1 \vee P_2)$, and $C_4 = (P_3)$ are part of a theory. Initially unit propagation during preprocessing assigns $P_1 = \text{FALSE}$, $P_3 = \text{TRUE}$, and $P_2 = \text{TRUE}$.

P_2 = TRUE is dependent on P_1 but P_3 is not. However, with the stacked-based algorithm, the stack at DL 0 simply contains the propositions P_1, P_2, and P_3 without any information on the dependency between these assignments. Therefore, if C_2 (or any other clause) is removed from the theory, all three propositions P_1, P_2, and P_3 must be unassigned, even though P_3 should still be assigned to TRUE without C_2.

3.2.2 Logic-based Truth Maintenance System

LTMS is an incremental unit propagation algorithm that can selectively unassign a single proposition and all its dependents, while leaving the rest of the assignments unchanged [6, 27]. Although LTMS refers to both the propagation and unassignment components of unit propagation, its key innovation lies within the unassign element, which uses clausal supports to identify dependents of a proposition.

The main ideas behind LTMS are detailed below. First, section 3.2.2.1 explains support in more detail and defines a well-founded support which is crucial for all TMS algorithms. Next, section 3.2.2.2 introduces the idea of how a proposition that has lost its supporting clause can be resupported by another clause. Finally, section 3.2.2.3 details the LTMS algorithm with unit propagation, variable unassignment, and resupport.

3.2.2.1 Well-founded Support

Supports are the backbone of truth maintenance systems and must be sound, i.e. well-founded, at all times. In plain words, a support C is the reason why supported proposition P holds its current assignment V. And it is important to ensure that this reason is valid before assigning P and remains valid while P = V. If a support becomes invalid, the supported proposition must be immediately unassigned.

C is a well-founded support for P =V if:

1. all other literals in C are FALSE,
2. the literal of P in C evaluates to TRUE, and
3. none of the other propositions in C depends on P.

The first two conditions are naturally satisfied by unit propagation, but may be violated during variable unassignment when a proposition in C becomes unassigned. If this happens, P will lose the reason for its assignment and must be unassigned as well. The third condition is more subtle and is designed to prevent loops in the support. For example, if $C_1 = (\neg P_1 \vee P_2)$ supports P_2 = TRUE and $C_2 = (\neg P_2 \vee P_1)$ supports P_1 = TRUE, then the two clauses form a loop support where P_1 depends on P_2 and P_2 depends on P_1.

Loops supports do not appear during unit propagation when all variables are initially unassigned. If P_1 and P_2 are unassigned, then neither C_1 nor C_2 could support either variable. One of P_1/P_2 must be assigned before C_1 or C_2 becomes unit, but then P_1/P_2 would be supported by some clause other than C_1 and C_2, and therefore, a loop would not be formed. However, if not careful, support loops can be introduced when resupporting a proposition.

3.2.2.2 Resupport

Resupport is built upon the idea that there are potentially multiple clauses that can provide well-founded support for proposition P. Therefore, if P's current support C is deleted from the theory, some other clause C' may still be able to support P's assignment, which means P does not need to be unassigned. However, this process of reassignment is complicated by the possibility of loop supports.

For example, in Figure 15, theory T_3 initially contains clauses $C_1 = (P_1)$ and $C_2 = (\neg P_1 \vee P_2)$. Unit propagation assigns P_1 to TRUE with C_1 as its support and P_2 to TRUE with C_2

as its support. Next, an incremental change to T_3 deletes P_1's support C_1 and adds clauses $C_3 = (\neg P_2 \vee P_1)$ and $C_4 = (P_2)$.

Unit propagation

$$\mathbf{T_3} = \quad \begin{array}{l} C_1 : (P_1) \wedge \\ C_2 : (\neg P_1 \vee P_2) \wedge \\ \ldots \end{array} \quad \Rightarrow \quad \begin{array}{l} C_1 : (P_1) \rightarrow P_1 = \text{TRUE} \wedge \\ C_2 : (\neg P_1 \vee P_2) \rightarrow P_2 = \text{TRUE} \wedge \\ \ldots \end{array}$$

$$\Downarrow$$

Context Switch

$$\mathbf{T_3' = T_3 - C_1 + C_3 + C_4 =} \quad \begin{array}{l} ?? \rightarrow P_1 = \text{TRUE} \\ C_2 : (\neg P_1 \vee P_2) \rightarrow P_2 = \text{TRUE} \wedge \\ C_3 : (\neg P_2 \vee P_1) \wedge \\ C_4 : (P_2) \wedge \\ \ldots \end{array}$$

$$\Downarrow$$

Unassign

$$\begin{array}{l} C_2 : (\neg P_1 \vee P_2) \wedge \\ C_3 : (\neg P_2 \vee P_1) \wedge \\ C_4 : (P_2) \wedge \\ \ldots \end{array}$$

$$\Downarrow$$

Resupport

$$\begin{array}{l} C_2 : (\neg P_1 \vee P_2) \wedge \\ C_3 : (\neg P_2 \vee P_1) \wedge \rightarrow P_1 = \text{TRUE} \\ C_4 : (P_2) \wedge \rightarrow P_2 = \text{TRUE} \\ \ldots \end{array}$$

Figure 15: Loop Support and Conservative Resupport Example

At first glance C_3 should be able to resupport P_1 because its literal P_1 evaluates to TRUE while the other literal $\neg P_2$ evaluates to FALSE. However, doing so will introduce a loop support between C_2 and C_3. Since the LTMS cannot identify loop supports, it employs a conservative resupport strategy that first unassigns P_1 and its dependent P_2 before resupporting P_2 with C_4 and P_1 with C_3.

3.2.2.3 Incremental Unit Propagation with Conservative Resupport

Figure 16 presents the pseudo-code for unit propagation with LTMS. It is identical to the unit propagation pseudo-code presented in Figure 9 except for the addition of line 10 where clause $C_P[i]$ is recorded as the support for a newly assigned proposition P_1.

```
propagate(P)
1       if P = TRUE
2               then C_P ← list of clauses with negative literals associated with P
3               else C_P ← list of clauses with positive literals associated with P
4       for i ← 1 to length(C_P)
5               do if C_P[i] is a unit clause
6                       then P_1 ← proposition in C_P[i] with value UNKNOWN
7                            if literal associated with P_1 is POSITIVE
8                              then P_1 ← TRUE
9                              else P_1 ← FALSE
10                           record C_P[i] as the support for P_1
11                           if propagate(P_1) = false
12                             then return false
13                  if C_P[i] is violated
14                      then return false
15      return true
```

Figure 16: LTMS Unit Propagation Pseudo-code

Figure 17 presents the pseudo-code used by LTMS to unassign a proposition and its dependents. The LTMS unassign algorithm is essentially the inverse of forward unit propagation. When a proposition P is assigned, unit propagation searches through the list of clauses containing P for unit or violated clauses. Likewise, when proposition P is unassigned, LTMS searches the list of clauses C_P containing P for any clause $C_P[i]$ that supports some other proposition P_1. Since P is unassigned, $C_P[i]$ can no longer provide support for P_1; therefore, P_1 must be unassigned (lines 5-6). For example, recall from section 3.2.1 that clause $C_1 = (\neg P_1 \vee P_2 \vee P_3)$ is part of some larger theory. P_1 was assigned TRUE by unit propagation of some other clause at DL 3, P_2 was assigned FALSE at DL 4, and P_3 was assigned TRUE by unit propagation of $C_1 = (\neg P_1 = \text{FALSE} \vee P_2 = \text{FALSE} \vee P_3)$ at DL 4; thus C_1 is the support for P_3. If the search is backtracking chronologically, unassigning the decision variable at DL 4 will lead to the unassignment of P_2, since P_2 results from unit propagation of that decision variable. When P_2 is unassigned, LTMS searches through clauses containing P_2. Among those clauses is C_1 which provides the support for P_3. Since P_2 was unassigned, C_1 can no longer support P_3, so P_3 is unassigned as well, and C_1 becomes ($\neg P_1 = \text{FALSE} \vee P_2 \vee P_3$).

```
unassign(P)
1     CP ← list of clauses with literals associated with P
2     P ← UNKNOWN
3     for i ← 1 to length(CP)
4             do if CP[i] supports some proposition P1
5                     then remove CP[i] as P1's support
6                          unassign(P1)
7                 if CP[i] is a unit clause
8                     then insert(CP[i], unitClauseList)
9     while not empty(unitClauseList)
10            do Cu←front(unitClauseList)
11                if Cu is a unit clause
12                    P←unassigned variable in Cu
13                    assign P so it's literal in Cu is TRUE
14                    propagate(P)
```

Figure 17: LTMS Unassign Pseudo-code

Furthermore, a LTMS can be used even if backtracking is not chronological. If the decision variable at DL 3 is unassigned when the decision variable at DL 4 is not, then P_1 will be unassigned while P_2 remains FALSE. However, when LTMS searches through clauses containing P_1, C_1 will still be identified as the support for P_3. Since, P_1 was unassigned, C_1 will no longer be able to support P_3, so P_3 will be unassigned and C_1 becomes ($\neg P_1 \vee P_2 = \text{FALSE} \vee P_3$).

While searching through clauses containing P during unassign(P) (line 3), LTMS also stores any unit clause C_u containing P into a list of unit clauses (lines 7-8); this unit clauses list is used to perform conservative resupport (lines 9-14) of P after P and its dependents are unassigned. Since P is unassigned, any unit clause C_u containing P must contain an unassigned literal associated with P while all other literals are FALSE. After P and all its dependents are unassigned, if C is still unit (line 11), then no other literal in C_u depends on P, and C_u can safely resupport P without introducing loop supports.

LTMS can incrementally perform changes to a theory where clauses are added and deleted in no specific order. Unlike the stacked-based approach that simply unassigns all propositions at DL 0, the LTMS saves computational effort by only unassigning propositions dependent on the deleted clauses. However, to do this a LTMS must search through clauses containing an unassigned variable, while a stacked-based approach need only unassign the value of a variable without worrying about its dependents. For example, clauses $C_2 = (\neg P_1)$, $C_3 = (P_1 \vee P_2)$ are part of a theory. Initially, unit propagation during preprocessing assigns P_1 = FALSE with C_2 as its support and then P_2 = TRUE with C_3 as the support. When a clause $C_4 = (P_3)$ is added to the theory, unit propagation will likewise assign P_3 = TRUE with C_4 as its support. If C_2 is deleted from the theory, P_1 that is supported by C_2 will be unassigned. Next, LTMS looks at clauses containing $P_1$and identifies C_3 as the support for P_2. Since P_1 is unassigned, P_2 loses its support and must be unassigned as well. C_4, on the other hand, does not contain P_1 or P_2 and will not be visited by unassign; therefore, P_3 will retain its assignment after C_2 is deleted.

3.3 Incremental Truth Maintenance System

As stated in section 3.2.2, the LTMS employs a conservative resupport strategy guaranteed to prevent loop supports. However, conservative resupport is not efficient because it could potentially unassign a large number of propositions that are soon reassigned the same values. For example, in Figure 18, either clauses C_1 and C_2 or clauses C_3 and C_4 can be propagated to support the assignment P_1 = TRUE; and propagating P_1 = TRUE leads to a large number of variable assignments. Arrows 1, 2, and 3 are used to indicate the supports and dependencies of variable assignments. Assume initially, clauses C_1 and C_2 are part of some larger theory T_4 and supports the assignments P_2 = TRUE and P_1 = TRUE (arrow 1) which leads to a large number of variable assignments (arrow 3). With conservative resupport, if a context switch deletes C_1 and C_2 from the theory while adding clauses C_3

and C_4, an LTMS will first unassign P_2, P_1 (delete arrow 1), and all their dependents (delete arrow 3), then propagate C_3 to assign P_3 to TRUE, propagate C_4 to resupport P_1 = TRUE (insert arrow 2), and finally reassign P_1's former dependents (reinsert arrow 3).

T_4 =

$C_1 : (P_2) \wedge$
$C_2 : (\neg P_2 \vee P_1) \wedge$
...

$T_4' = T_4 - C_1 - C_2 + C_3 + C_4$ =

$C_3 : (P_3) \wedge$
$C_4 : (\neg P_3 \vee P_1) \wedge$
...

$$\begin{tikzcd} T_4 \arrow[dr, "1"'] & & T_4' \arrow[dl, "2"] \\ & P_1 = \text{TRUE} \arrow[d, "3"'] & \\ & \text{[Large number of assignments dependent on } P_1\text{]} & \end{tikzcd}$$

Figure 18: Conservative vs Aggressive Resupport Example

In this example, the values of P_1 and its dependents are the same before and after the context switch. Therefore, it is desirable to avoid unassigning and reassigning them during the context switch.

The ITMS [22] is a derivative of LTMS used to reduce these unnecessary changes to variable assignments. We use this algorithm as a building block for the ITMS with watch-literals algorithm used in Chapter 4.

Unlike the LTMS whose key innovation lies in its variable unassignment algorithm, an ITMS makes significant alterations to both the forward propagation and backward unassignment components of unit propagation. During a context switch, an ITMS first propagates newly added clauses before unassigning propositions supported by the deleted clauses; this increases the chance of resupporting a proposition. It also employs an aggressive resupport strategy that immediately ressuports a variable assignment (if possible) without unassigning its dependents. For the same example in Figure 18, during the context switch, an ITMS will first propagate C_3 to assign P_3 to TRUE. At this point, C_4

$= (\neg P_3 = \text{FALSE} \vee P_1 = \text{TRUE})$ could provide well-founded support for $P_1 = \text{TRUE}$ if needed; however, since P_1 is already supported by C_2, propagation terminates. When C_1 and C_2 are removed from the theory, the algorithm removes C_2 as P_1's support (delete arrow 1) and searches for a new support. Since C_4 can provide well-founded support for P_1, it is set as P_1's new support (insert arrow 2). During this process, the values of P_1 and its dependents remain unchanged.

There are two difficulties faced by the propagate-before-unassign and aggressive resupport algorithms: loop supports and mutual inconsistencies between added and deleted clauses. Sections 3.3.1 and 3.3.2 detail these problems and their solutions, called propagation numbering and conflict repair respectively; and section 3.3.3 contains the detailed algorithm for aggressive resupport.

3.3.1 Propagation Numbering

Recall from section 3.2.2 that directly resupporting a proposition without unassigning its dependents could lead to loop supports. In order to resolve this problem, the ITMS introduces a depth-first numbering scheme for propagation assignments. Each proposition has an associated propagation number, N_P, whose value is determined by the following rules:

1. For unassigned propositions, $N_P = 0$,
2. For decision variables, $N_P = 1$,
3. And for propositions whose assignment is supported by a clause C, $N_P \geq 1 +$ max(propagation numbers of other propositions in C)

If the assignment of P_2 is dependent on P_1 then P_2's propagation number must be larger then P_1's. Therefore, as long as these rules are observed, P_2 can never provide support for P_1, thus avoiding the possibility of loop supports.

3.3.2 Conflict Repair

As stated earlier, in order to maximize the chances of resupporting a proposition, an ITMS first propagates newly added clauses before unassigning propositions supported by deleted clauses. However, if the new and deleted clauses are mutually inconsistent then propagating the new clauses using unit propagation will lead to conflicts that would normally terminate propagation. For example, T_5 in Figure 19 is the same as T_4 in Figure 18 except for the addition of C_5. Initially, unit propagation of C_1, C_2, and C_5 leads to the assignments P_2 = TRUE, P_1 = TRUE, (arrow 1) and the assignment of its dependents (arrow 3), and P_3 = FALSE. Next, a context switch adds clauses C_3 and C_4 while removing clauses C_1, C_2, and C_5 so that $T_5' = T_5 - C_1 - C_2 + C_3 + C_4 - C_5$. Here, without unassigning P_3, $C_3 = (P_3 = \text{FALSE})$ is violated thus terminating unit propagation. Since $C_4 = (\neg P_3 = \text{TRUE} \vee P_1 = \text{TRUE})$ cannot provide well-founded support for P_1, when C_2 is deleted, P_1 and its dependents must be unassigned (delete arrows 1 and 3). When C_5 is deleted and P_3 is unassigned, C_3 becomes a unit clause and is propagated to support P_3 = TRUE. Next, C_4 is propagated to support P_1 = TRUE (insert arrow 2). At this point, unit propagation can reassign all former dependents of P_1 (reinsert arrow 3).

$\mathbf{T_5 =}$	$\mathbf{T_5' = T_5 - C_1 - C_2 + C_3 + C_4 - C_5 =}$
$C_1 : (P_2) \wedge$	$C_3 : (P_3) \wedge$
$C_2 : (\neg P_2 \vee P_1) \wedge$	$C_4 : (\neg P_3 \vee P_1) \wedge$
$C_5 : (\neg P_2 \vee \neg P_3) \wedge$	…
…	

1 ↘ ↙ **2**

P_1 = TRUE

3 ↓

[Large number of assignments dependent on P_1]

Figure 19: Mutual Inconsistency Example

In this example, the value of P_1 and its dependents are assigned the same values before and after the context switch. However, due to the mutual inconsistency between C_1, C_5, and C_3, P_1 could not be resupported aggressively leading to the unassignment and reassignment of a large number of variables.

In order to circumvent this problem and fully propagate added clauses prior to deletion, the ITMS introduces a conflict repair technique that allows for propagation of not only unit but violated clauses as well. Figure 20 contains pseudo-code for propagation with ITMS, and Figure 21 presents the pseudo-code for conflict repair.

```
propagate(P)
1    if P = TRUE
2        then C_P^T ← list of clauses with positive literals associated with P
3             C_P^F ← list of clauses with negative literals associated with P
4        else C_P^F ← list of clauses with positive literals associated with P
5             C_P^T ← list of clauses with negative literals associated with P
6    noConflict ← true
7    for i ← 1 to length(C_P^F)
8        do if C_P^F[i] is a unit clause
9            then P_1 ← proposition in C_P^F[i] with value UNKNOWN
10               if literal associated with P_1 is POSITIVE
11                 then P_1 ← TRUE
12                 else P_1 ← FALSE
13               record C_P^F[i] as the support for P_1
14               N_P1 ← 1 + max(propagation # of other propositions in C_P^F[i])
15               if propagate(P_1) = false
16                 then noConflict ← false
17          if C_P^F[i] is violated
18            then if not repairConflict(C_P^F[i])
19                 then noConflict ← false
20   if (P has been flipped)
21       then for j ← 1 to length(C_P^T)
22           do if C_P^T[j] supports a proposition
23                 then P_1 ← proposition supported by C_P^T[j]
24                      unassign(P_1)
25   return noConflict
```

Figure 20: ITMS Propagation Pseudo-code

When a violated clause C is encountered (Figure 20 line 17), the ITMS attempts to repair C by flipping the truth assignment V of a proposition P in C so that P's literal evaluates to

TRUE after the assignment change (Figure 21 lines 3-5). After flipping P's value from V to ¬V, C becomes P's new support (Figure 21 line 7), and P's propagation number is changed to 1 + max(propagation numbers of other propositions in C) (Figure 21 line 6). Once P's assignment changes, propositions dependent on P = V are unassigned (Figure 20 lines 20-24) and the assignment P = ¬V is propagated (Figure 20 lines 7-19).

```
repairConflict(C)
1       P ← proposition in C with largest propagation number
2       if P has not been flipped before
3               then if P = TRUE
4                       then P ← FALSE
5                       else P ← TRUE
6                    N_P ← 1+ max(propagation # of other propositions in C)
7                    record C as the support for P
8                    if (propagate(P) = false)
9                       then return false
10                   return true
11              else return false
```

Figure 21: ITMS Conflict Repair Pseudo-code

P must meet the following conditions for conflict repair to take place:

1. P has the highest propagation number in C; ties are allowed (Figure 21 line 1);
2. P's value has not been flipped during this context switch (Figure 21 line 2).

The first condition ensures that supporting P with C does not introduce a loop support. With the propagation numbering system, if P' is dependent on P, then its propagation number must be larger than that of P's. Thus, if all other propositions in C have propagation numbers less than or equal to P's, then none of those propositions are dependent on P, thus eliminating the possibility of a loop support. Furthermore, a tie in propagation numbers does not prevent C from supporting P because P's propagation number is updated after its assignment change.

The second condition prevents the algorithm from falling into an infinite loop where the same proposition is flipped back and forth. Since a proposition can only be repaired once per context switch, C cannot be repaired if P has already been flipped. In this case conflict repair returns false (Figure 21 lines 2, 11 and Figure 20 lines 17-18), and propagation terminates.

For the example in Figure 19, initially P_1 = TRUE with $N_{P1} = 2$, P_2 = TRUE with $N_{P2} = 1$, and P_3 = FALSE with $N_{P2} = 2$. When $C_3 = (P_3 = \text{FALSE})$ is identified as a violated clause during the context switch, propagation with conflict repair will repair C_3 by flipping the value of P_3 and reset N_{P2} to 1—since there are no other propositions in C_3 and P_3 has not been flipped, the conditions for repair are satisfied. At this point, $C_4 = (\neg P_3 = \text{FALSE} \vee P_1 = \text{TRUE})$ could provide well-founded support for P_1 = TRUE because $N_{P1} > N_{P3}$; however, since P_1 is supported by C_2, propagation terminates. When C_1 is removed from the theory, the aggressive resupport algorithm can then remove C_2 as P_1's support (delete arrow 1) and resupport P_1 with C_4 (insert arrow 2). During this process, the values of P_1 and its dependents remain unchanged.

3.3.3 Aggressive Resupport

Recall aggressive resupport is used by the ITMS during retraction of variable assignments to minimize the number of unnecessary unassignments. When a proposition, P, losses its support, the ITMS looks for an alternative support for P and only unassigns P if such a support does not exist. A clause, C, can provide well-founded support for P if it meets the following conditions:

1. All other literals in C evaluate to false.
2. P appears with the same polarity (positive or negative) in both C and its old support.
3. P has the single largest propagation number in C; no ties allowed.

The first condition simply checks that C is capable of supporting variable P, while the second condition ensures that P's literal evaluates to TRUE in C and thus P's value will not be changed with C as its new support. Recall, for both unit propagation and conflict repair, if C supports P then P's literal is the single satisfying literal in C. Therefore, if P appears in the same polarity in both C and its old support, then its literal must evaluate to TRUE in both cases.

The third condition is enforced to prevent loops supports. In this case, a tie for the maximum propagation number in C is not allowed because a series of such ties could lead to a loop support. For example, proposition P_1 = TRUE with $N_{P1} = 3$, and proposition P_2 = FALSE with $N_{P3} = 3$. If ties are allowed for the third condition of resupport, then when P_1 losses its support, clause $C_1 = (P_2 = \text{FALSE} \vee P_1 = \text{TRUE})$ can be used as its new support. And when P_2 losses its support, clause $C_2 = (\neg P_1 = \text{FALSE} \vee \neg P_2 = \text{TRUE})$ can likewise be used to resupport P_2. However, when this happens, the supports are no longer well-founded because clauses C_1 and C_2 form a loop support.

```
unassign(P)
    C_P ← list of clauses with literals associated with P
    P ← UNKNOWN
    N_P ← 0
    for i ← 1 to length(C_P)
        do if C_P[i] supports some proposition P_1
            then remove C_P[i] as P_1's support
                if not resupport(P_1)
                    then unassign(P_1)
```

Figure 22: ITMS Unassign Pseudo-code

```
resupport(P)
    if P = TRUE
        then C_P ← list of clauses with positive literals associated with P
        else C_P ← list of clauses with negative literals associated with P
    for j ← 1 to length(C_P)
        do if all other literals in C_P[j] are FALSE and
              N_P > propagation number of all other propositions in clause
            then set C_P as P's new support
                return true
    return false
```

Figure 23: ITMS Aggressive Resupport Pseudo-code

Figure 22 and Figure 23 presents ITMS's unassign and aggressive resupport pseudo-codes. Since both P's value and its propagation number N_P remains unchanged during resupport, propagation numbers and values of P's dependents also do not need to be altered.

3.4 Root Antecedent ITMS

The propagation numbering system is an effective way to ensure soundness of supports. However, its conditions (defined in section 3.3.1) are sufficient but not necessary to avoid loop supports. If P_2's value depends on P_1 then its propagation number must be larger than P_1's; however, just because P_2 has a larger propagation number does not mean it depends on P_1. Therefore, although the propagation numbering system ensures soundness, it may overlook valid repair and resupport opportunities.

$\mathbf{T_6 =}$

$C_1 : (P_1) \wedge$
...

$\mathbf{T_6' = T_6 - C_1 + C_3 + C_4 =}$

$C_3 : (P_3) \wedge$
$C_4 : (\neg P_3 \vee P_1) \wedge$
...

1 ↘ ↙ **2**

P_1 = TRUE

3 ↓

[Large number of assignments dependent on P_1]

Figure 24: Propagation Numbering vs. Root Antecedent example

For example, in Figure 24, C_1 is part of some larger theory T_6 and supports the assignment P_1 = TRUE with $N_{P1} = 1$ (arrow 1), which leads to the assignment of a large number of variables (arrow 3). When clauses C_3 and C_4 are added to the theory while clause C_1 is deleted, an ITMS first propagates C_3 to assign P_3 to TRUE with $N_{P3} = 1$. However, $C_4 = (\neg P_3 = \text{FALSE} \vee P_1 = \text{TRUE})$ is not a valid support under the propagation numbering system because N_{P3} is the same as N_{P1} not smaller, even though P_3 does not depend on P_1. Therefore, when C_1 is deleted, P_1 and all its dependents are unassigned. When this happens, C_4 becomes unit and can be propagated to resupport P_1 = TRUE (arrow 2), finally, all of P_1's former dependents are reassigned (arrow 3).

Also, with the propagation numbering system, there are situations where a resupported proposition P losses its support after clauses are deleted in a context switch. For example, clause $C_1 = (P_1)$ supports P_1 = TRUE with $N_{P1} = 1$, $C_2 = (\neg P_1 \vee P_2)$ supports P_2 = TRUE with $N_{P2} = 2$, and $C_3 = (P_3)$ supports P_3 = TRUE with $N_{P3} = 1$. A context switch adds clause $C_4 = (\neg P_3 \vee P_2)$ and deletes clauses C_1, C_2, and C_3. Since C_4 is not unit or violated, propagation does not take place. When C_1 and C_2 are deleted, an ITMS using propagation numbers will resupport P_2 with $C_4 = (\neg P_3 = \text{FALSE} \vee P_2 = \text{TRUE})$. Since $N_{P3} < N_{P2}$, C_4 is a well-founded support for P_2. However, when C_3 is deleted, P_3 becomes unassigned, and C_4 can no longer support P_2. Therefore, P_2 must be unassigned. It is inefficient to resupport P_2 only to have P_2 immediately unassigned afterwards.

Root Antecedent ITMS (RA-ITMS) [27] introduces an alternative to propagation numbers called root antecedents. Root antecedents allow the algorithm to determine dependencies between propositions without any approximation, therefore increasing the chances of repair and resupport. Under the propagation numbering system, all root antecedents would have propagation numbers of 1. These are the propositions that do not depend on any other proposition for their value: i.e. decision variables or propositions supported by single literal clauses.

The root antecedent list R_P for proposition P contains all root antecedents in the theory that P depends on and can be used to determine the dependency between P and another proposition P' in the following way:

1. P depends on P' if $R_P \supseteq R_{P'}$,
2. P' depends on P if $R_P \subseteq R_{P'}$,
3. else P and P' does not depend on each other.

For the example in Figure 24, C_1 supports P_1 = TRUE with $R_{P1} = \{P_1\}$ (arrow 1), which led to the assignment of a large number of variables (arrow 3). When clauses C_3 and C_4 are added to the theory, and clause C_1 is deleted, RA-ITMS propagates C_3 to assign P_3 to TRUE with $R_{P3} = \{P_3\}$. However, unlike with propagation numbers, $C_4 = (\neg P_3 = \text{FALSE} \vee P_1 = \text{TRUE})$ can be used to resupport P_1 using the root antecedent system because $R_{P1} \not\subset R_{P3}$ indicating P_3 does not depend on P_1. But since P_1 is supported by C_1, propagation terminates. When C_1 is deleted from the theory, P_1 loses its support (delete arrow 1). RA-ITMS resupports P_1 with C_4 and R_{P1} becomes $\{P_3\}$ (insert arrow 2). Since P_1 is resupported, the values of P_1's dependents do not need to be altered. However, the root antecedent system does require updates to the root antecedent lists of all dependents of P_1 to reflect the change in R_{P1}: i.e. remove P_1 and insert P_3 from their root antecedent lists.

Also, root antecedents can be used to avoid repairing or resupporting with clauses containing propositions dependent on deleted clauses. For example, clause $C_1 = (P_1)$ supports P_1 = TRUE with $N_{P1} = 1$, $C_2 = (\neg P_1 \vee P_2)$ supports P_2 = TRUE with $N_{P2} = 2$, and $C_3 = (P_3)$ supports P_3 = TRUE with $N_{P3} = 1$. A context switch adds clause $C_4 = (\neg P_3 \vee P_2)$ and deletes clauses C_1, C_2, and C_3. Since C_4 is not a unit clause, propagation does not take place. However, with RA-ITMS C_4 is not a valid resupport for P_2 because $R_{P3} = \{P_3\}$, and P_3 is supported by deleted clause C_3. Therefore, when C_1 and C_2 are deleted, P_2 cannot be resupported and is directly unassigned.

Although root antecedent system can be used to more precisely determine dependencies between propositions, it also has some major drawbacks compared to the propagation numbering scheme. First of all, a roots antecedent list can potentially contain a large number of propositions and, therefore, demands more memory than a single propagation number: this memory must be allocated, released, and garbage collected dynamically. Also, while conflict repair and aggressive resupport with propagation numbers are solely based on the propagation depth of a variable, the root antecedent algorithm must identify and reference specific propositions affected by the context switch. Finally, a key difference between the two ITMS algorithms is that the RA-ITMS must update the root antecedent lists of all propositions that depend on a resupported proposition P, while the original ITMS requires no such changes to the propagation numbers.

Given these drawbacks and the lack of empirical results to support its performance, RA-ITMS is not used in the new algorithms presented in section 4. However, its ideas are similar to a decision-level ITMS with watched-literals algorithm (DL-ITMS-WL) that we designed. Although we are unable to implement and test the DL-ITMS-WL in time for this thesis, its main concepts are outlined in Chapter 7

The details of RA-ITMS are presented below. Section 3.4.1 lists the rules used to update a root antecedent list; and sections 3.4.2 and 3.4.3 incorporate root antecedents into the conflict repair and aggressive resupport algorithms, respectively.

3.4.1 Root Antecedents

As mentioned earlier, instead of propagation numbers, RA-ITMS associates with each proposition a list of propositions called its root antecedents. The root antecedent list, R_P, for a proposition, P, is updated by the following rules:

1. For unassigned propositions, $R_P = \{\}$.
2. For decision variables and propositions supported by a single literal clause, $R_P = \{P\}$.
3. For propositions assigned through propagation of clause C, R_P = union(root antecedents of all other propositions in the C).

3.4.2 Conflict Repair

Root antecedents can be used by the conflict repair algorithm to replace propagation numbers, while leaving the rest of the algorithm unchanged. To do this, the algorithm must not only consider the root antecedents of propositions in a violated clause but also the propositions supported by the deleted clause(s). For example, a context switch includes the addition of clause $C_1 = (P_1)$ and deletion of clause $C_2 = (P_2)$. If propagation of C_2 leads to the violation of clause C, then the assignment of a proposition P can be flipped to repair C if P is the only proposition in C with $P_2 \subseteq R_P$.

This condition ensures that:

1. the conflict leading to C's violation is caused by the context switch,
2. P will not lose C as its support after the context switch,
3. P does not depend on any other proposition in C, and
4. P has not been flipped during this context switch.

Assume $C = (P' = FALSE \vee P = FALSE)$. If P_2 is part of neither R_P nor $R_{P'}$, then C's violation is not due to mutual inconsistencies within the context switch and would remain violated even after C_2 is deleted; therefore, C should not be repaired. If $P_2 \subseteq R_P$ and $P_2 \subseteq R_{P'}$ and P is flipped with C as its new support, then when P_2 is unassigned, P', whose value is dependent on P_2 will also be unassigned, causing P to lose its support. If $P_2 \subseteq R_P$ and $P_2 \not\subset R_{P'}$, then P' is not dependent on P because $R_P \not\subset R_{P'}$; therefore, C can provide well-founded support for P = TRUE. After the value of P is flipped, C becomes P's support and $R_P = R_{P'}$ so $P_2 \not\subset R_{P'}$. Since P's new support cannot contain another proposition with P_2 as a root antecedent, P cannot depend on P_2 after the assignment change. Inversely, if P depends on P_2, then its assignment has not been flipped during this context switch.

Figure 25 and Figure 26 contains the pseudo-code for the propagation and conflicts repair algorithms in RA-ITMS. The only difference between them and the propagate and conflict repair pseudo-codes for ITMS (*see section 3.3.2*) are in Figure 25 line 13 and Figure 26 lines 1 and 5.

```
propagate(P)
    if P = TRUE
        then C_P^T ← list of clauses with positive literals associated with P
             C_P^F ← list of clauses with negative literals associated with P
        else C_P^F ← list of clauses with positive literals associated with P
             C_P^T ← list of clauses with negative literals associated with P
    for i ← 1 to length(C_P^F)
        do if C_P^F[i] is a unit clause
            then P_1 ← proposition in C_P^F[i] with value UNKNOWN
                if literal associated with P_1 is POSITIVE
                    then P_1 ← TRUE
                    else P_1 ← FALSE
                record C_P^F[i] as the support for P_1
                R_P ← union (RA of other propositions in C_P^F[i])
                if propagate(P_1) = false
                    then return false
           if C_P^F[i] is violated
            then if not repairConflict(C_P^F[i])
                    then return false
    if (P has been flipped)
        then for j ← 1 to length(C_P^T)
            do if C_P^T[j] supports a proposition
                then P_1 ← proposition supported by C_P^T[j]
                    unassign(P_1)
    return true
```

Figure 25: RA-ITMS Propagation Psedo-code

```
repairConflict(C)
    if C contains one and only one proposition P dependent on a deleted clause
        then if P = TRUE
                then P ← FALSE
                else P ← TRUE
             R_P ← union(root antecedent of other propositions in clause)
             if propagate(P) = false
                then return false
             return true
        else return false
```

Figure 26: RA-ITMS Conflict Repair Pseudo-code

3.4.3 Aggressive Resupport

Root antecedents can also be used by the aggressive resupport algorithm to replace propagation numbers. Like conflict repair, aggressive resupport with root antecedents must consider root antecedents of propositions in the resupporting clause along with propositions supported by the deleted clause(s). For example, a context switch includes the addition of clause $C_1 = (P_1)$ and deletion of clause $C_2 = (P_2)$. If unassignment of P_2 causes a proposition P to lose its support, then a clause C can resupport P if it meets the following conditions:

1. All other literals in C evaluate to false;
2. P appears with the same polarity (positive or negative) in both C and its old support;
3. P is the only proposition in C with $P_2 \subseteq R_P$.

The first two conditions are the same as those in section 3.3.3 and simply ensure that C can support P's current assignment. The third condition ensures that no other propositions in C depend on P. Recall, P' depends on P if $R_P \subseteq R_{P'}$; since $P_2 \subseteq R_P$ and P_2 is not in the root antecedent lists of other propositions of C, those propositions cannot depend on P. Therefore, C is a well-founded support for P.

```
resupport(P)
    if P = TRUE
        then C_P ← list of clauses with positive literals associated with P
        else C_P ← list of clauses with negative literals associated with P
    for j ← 1 to length(C_P)
        do if all other literals in C_P[j] are FALSE and
              P is only proposition in C_P[j] dependent on a deleted clause
                then set C_P as P's new support
                     R_P ← union(root antecedent of other propositions C)
                     Update root antecedents of all propositions dependent on P
                     return true
    return false
```

Figure 27: RA-ITMS Aggressive Resupport Pseudo-code

Figure 27 presents the aggressive resupport pseudo-code for RA-ITMS. The RA-ITMS unassign function is the same as ITMS (*see section 3.3.3 Figure 22*).

3.5 Summary

Although the concept of unit propagation is simple, there are many variations to its implementation that can greatly affect its performance. Chapter 3 presented an overview of these techniques, which can be broken into two categories. One focuses on the workload required for a SAT solver to identify unit and violated clauses through the use of different data structures. The other concentrates on the number of propositions assigned and reassigned during clause addition, clause deletion, and backtrack search.

The three data structures discussed in this chapter are the counter-based approach, head/tail lists, and watched-literals. Among these, the counter-based approach belongs to a group of data structures called the adjacency list. Although their details may vary, adjacency list data structures all require the unit propagation algorithm to search through the list of all clauses associated with a newly assigned proposition. Lazy data structures, on the other hand, allow the algorithm to only searches through a subset of these clauses. These data structures, such as the head/tail lists and watched-literals, that place special emphasis on two literals per clause, have an average saving of 1:average-number-of-literals-per-clause over adjacency lists.

Within lazy data structures, head/tail lists is an earlier algorithm that requires updates to a clause's head and tail literals during variable unassignment. In comparison, the watched-literals scheme has the performance advantage because the two watched literals can remain unchanged during chronological backtracking. Empirical results confirm that watched-literals performs better than head/tail lists over a variety of SAT instances [14].

There are also two subtypes among those algorithms affecting of the number of propositions assigned and unassigned during clause addition, deletion, and search. The first subtype, which includes stack-based backtracking and the LTMS, is primarily used to determine how variables are unassigned. Stack-based backtracking is a non-incremental algorithm that unassigns entire decision levels during search backtracking and all assignments when the theory is altered. The LTMS is an incremental algorithm that can selectively unassign a proposition and its dependents; this approach avoids unassigning and reassigning propositions whose supporting clauses are not affected by the change to the theory.

The second subtype includes two ITMS algorithms. Although incremental, they are different than the LTMS in that an ITMS is specifically designed to minimize variable unassignments during context switches that involve both the addition and deletion of clauses. The ITMS algorithms use the conflict repair technique to propagate added clause before unassigning propositions supported by the deleted clauses. They also search for alternative supports for propositions that lost their supports and only unassign variables that cannot be resupported.

The key challenge to implementing aggressive resupport is the possibility of forming loops in the supports. The ITMS uses depth-first numbering system called propagation numbers prevent these loop supports. Empirical results show that the ITMS is more efficient than the LTMS at minimizing variable unassignments and reassignments across a series of context switches [22].

The RA-ITMS, on the other hand, uses root antecedent lists instead of propagation numbers. Although more precise at identifying dependency between propositions, root antecedents require more work than propagation numbers in the form of updates to dependents of resupported propositions. And there lacks empirical evidence to support any performance gain of an RA-ITMS over an ITMS.

Since the watched-literals scheme is the best performing SAT data structure to-date, and the LTMS and the ITMS are two efficient incremental unit propagation algorithms with the latter targeting specifically at context switches, it is desirable to implement the LTMS and the ITMS with the watched-literals data structure in order to optimize SAT performance. Chapter 4 details the challenge in using watched-literals with the TMS algorithms, our solution approach, and the new LTMS with watched-literals and ITMS with watched-literals algorithms.

Chapter 4

Truth Maintenance with watched literals

In Chapter 4, we deliver two new incremental unit propagation algorithms called logic-based truth maintenance system with watched-literals (LTMS-WL) and incremental truth maintenance system with watched-literals (ITMS-WL). These algorithms combine the strengths of the watched-literals data structure with the LTMS and ITMS algorithms in order to improve the performance of unit propagation in SAT solvers.

There is one key challenge that arises from the use of watched-literals data structure with the TMS algorithms. Recall, from section 3.1.3 that watched-literals is one of the most efficient SAT data structures that allows for efficient unit propagation by visiting only a subset of the clauses associated with a newly assigned proposition. One advantage of the algorithm is that watched literals do not need to be updated when unassignment takes places in reverse order of assignment. The logic-based and incremental truth maintenance systems, on the other hand, improve unit propagation efficiency by reducing the number of unnecessary variable unassignments and reassignments during context switches (*see sections 3.2.2 and 3.3*). These algorithms selectively unassign a proposition and all its dependents while leaving other assignments in place regardless of the order in which propositions were assigned. Since the TMS algorithms do not necessarily unassign variables in order, watched literals may need to be updated under certain situations.

Recall from section 3.1.3 that when variables are not unassigned in order, some clauses may contain unknown literals even through one or both of its watched literals are FALSE. For example, clause C_1 = (**P_1 = FALSE** $\vee$ $\neg P_2$ = FALSE $\vee$ **P_3 = FALSE** $\vee$ $\neg P_4$ = FALSE).

If only P_4 is unassigned, then the value of unwatched literal $\neg P_4$ in C_1 will be unknown while watched literals P_1 and P_3 are FALSE. When this happens, the watched literals for C_1 must be updated so that $\neg P_4$ is watched and either P_1 or P_3 unwatched.

However, in order to identify C_1 through unwatched literal $\neg P_4$ during variable unassignment, the algorithm must search through the list of all clauses containing literals associated with P_4, not just those containing watched literals. But since the list of watched literals is a subset of all literals, we would like to use the watched literals list whenever possible, and only consider the list of all literals when absolutely necessary. Therefore, our solution approach associates with each proposition its lists of watch and all literals and selectively uses these lists depending on the situation. Algorithmic details for LTMS-WL and ITMS-WL are explored in the sections 4.1 and 4.2, respectively.

4.1 LTMS with watched-literals

This section applies the watched-literals data structure to the LTMS algorithm introduced in section 3.2.2.

Using watched literals with the forward propagation component of LTMS is simple. The LTMS-WL unit propagation pseudo-code in Figure 28 is just a merge of the watched-literals and LTMS pseudo codes found in Figure 12 and Figure 16 in sections 3.1.3 and 3.2.2. Like in section 3.1.3, when a proposition P is assigned, propagate searches through the list of clauses C_P containing either positive or negative watched literals associated with P depending on P's value (Figure 28 lines 1-3). For each clause $C_P[i]$, the algorithm attempts to replace the FALSE watched literal, W^1, associated with P with a non-false, unwatched literal if possible (Figure 28 line 8). Otherwise, $C_P[i]$ is unit if its other watched literal W^2 is not assigned (Figure 28 line 9) or violated if W^2 is FALSE (Figure 28 line 17). If $C_P[i]$ is unit, then the proposition P_1 associated with W^2 is assigned such that W^2 evaluates to TRUE (Figure 28 lines 11-13). And, like in section 3.2.2, $C_P[i]$ is recorded as

P_1's support. Propagation terminates when there are no more unit clauses or when a conflict is encountered.

```
propagate(P)
    if P = TRUE
        then C_P ← list of clauses with negative watched literals associated with P
        else C_P ← list of clauses with positive watched literals associated with P
    noConflict ← true
    for i ← 1 to length(C_P)
        do W^1 ← watched literal in C_P[i] associated with P
           W^2 ← other wathed literal in C_P[i]
           replace W^1 with non-FALSE, unwatched literal if possible
           if W^1 cannot be replaced and W^2 = UNKNOWN
               then P_1 ← proposition associated with W^2
                    if W^2 is POSITIVE
                       then P_1 ← TRUE
                       else P_1 ← FALSE
                    record C_P[i] as the support for P_1
                    if propagate(P_1) = false
                       then noConflict ← false
           if W^1 cannot be replaced and W^2 = FALSE
               then noConflict ← false
    return noConflict
```

Figure 28: LTMS-WL Unit Propagation Pseudo-code

Since there are only 2 watched literals per clause, watched literals lists are on average 2/avg-#-of-literals-per-clause shorter than adjacency lists containing all literals associated with a proposition. Furthermore, during propagation of proposition P, clauses containing TRUE literals of P do not need to be updated because TRUE literals can be watched regardless of the values of the unwatched literals. Therefore, clauses visited by LTMS-WL's propagation algorithm is 1/avg-#-of-literals-per-clause less than an LTMS using adjacency list data structures such as the counter-based method.

The LTMS-WL unassign algorithm is slightly more complicated and can be divided into two versions. One is used within chronological backtrack search while the other is used during preprocessing where unassignments are not necessarily in order. The details of these algorithms are presented in sections 4.1.1 and 4.1.2 below.

4.1.1 LTMS-WL for Chronological Backtracking

The main purpose of the LTMS-WL unassign algorithm is twofold. One is to ensure that the set of supports remain well-founded by identifying and unassigning all dependents of unassigned propositions. The other is to maintain the integrity of the watched literals data structure so that propagate can accurately identify unit and violated clauses. Within chronological backtracking, these tasks can be accomplished by searching through clauses containing only FALSE watched literals associated with an unassigned proposition. Like with LTMS-WL's propagate algorithm, the use of watched literals in LTMS-WL unassign for chronological backtracking reduces the number of clauses visited by a factor of 1/avg-#-of-literals-per-clause compared to adjacency lists.

For an example of how this algorithm works, consider clauses:

$C_1 = (\mathbf{\neg P_1 = FALSE} \vee \mathbf{P_2 = TRUE})$,
$C_2 = (\neg P_1 = \text{FALSE} \vee \mathbf{\neg P_2 = FALSE} \vee \mathbf{P_3 = TRUE})$, and
$C_3 = (\neg P_1 = \text{FALSE} \vee \mathbf{\neg P_4 = FALSE} \vee \mathbf{P_5 = TRUE})$.

P_1 = TRUE and P_4 = TRUE are decision variables; and P_2 = TRUE, P_3 = TRUE, and P_5 = TRUE are supported by clauses C_1, C_2, and C_3 respectively. These propositions are assigned in the order P_1, P_2, P_3, P_4, and P_5. Since chronological backtracking refers to the chronological unassignment of decision variables, the algorithm first unassigns the last assigned decision variable P_4. When searching through the list of clauses containing FALSE watched literals of P_4, C_3 is identified through watched literal $\neg P_4$. With P_4 unassigned, C_3 can no longer support P_5 = TRUE; therefore, P_5 is unassigned as well.

Next, decision variable P_1 is unassigned. When searching through the list of clauses containing FALSE watched literals of P_1, C_1 is identified through watched literal $\neg P_1$. With P_1 unassigned, C_1 can no longer support P_2 = TRUE; therefore, P_2 is unassigned, which then leads to the identification of clause C_2 and the unassignment of proposition P_3, at which point all propositions are unassigned.

Since literal $\neg P_1$ is not watched in C_2, C_2 was not visited when P_1 was unassigned even though P_3 = TRUE depended on P_1 as well as P_2. However, unassignment of P_1 led to the unassignment of P_2 which in turn led to the unassignment of P_3. So the set of supports remain well founded by the end of unassign.

```
unassign(P)
    if P = TRUE
        then C_P ← list of clauses with positive watched literals associated with P
        else C_P ← list of clauses with negative watched literals associated with P
    P ← UNKNOWN
    for i ← 1 to length(C_P)
        do if C_P[i] supports some proposition P_1
            then remove C_P[i] as P_1's support
                unassign(P_1)
          if C_P[i] is a unit clause
            then insert(C_P[i], unitClauseList)
    while not empty(unitClauseList)
        do C_u←front(unitClauseList)
          if C_u is a unit clause
            P←unassigned variable in C_u
            assign P so it's literal in C_u is TRUE
            propagate(P)
```
Figure 29: LTMS-WL Unassign Pseudo-code for Chronological Backtracking

In general, for clause C = (U_1 = FALSE $\vee$ $\mathbf{W_1}$ **= FALSE** $\vee$ $\mathbf{W_2}$ **= TRUE**) where C supports proposition W_2, U_1 must have been assigned before W_1 and W_2 because supported literal W_2 must be the last literal assigned and FALSE watched literal W_1 can only remain watched if other unwatched literals are already FALSE. Therefore, if U_1 is dependent on decision variable P_D, then W_1 and W_2 must be dependent P_D or some other decision variable assigned after P_D. Since decision variables are unassigned in order, if U_1 becomes unassigned, then W_1 and thus W_2 must be unassigned as well, even if that unassignment is not directly triggered by U_1. This ensures the soundness of the set of supports. Also, since any unassignment of unwatched literal U_1 will be followed by the unassignment of watched literals W_1 and W_2, the watched literals do not need to be updated during chronological backtracking.

Figure 29 presents the pseudo-code for LTMS-WL unassign. When a variable P needs to be unassigned, LTMS-WL searches through the list of clauses C_P containing P's watched literals that evaluates to FALSE (Figure 29 lines 1-3). Once P becomes unassigned (Figure 29 line 3), clauses containing P can no longer provide support for other propositions; therefore, if clause $C_P[i]$ supports some proposition P_1, then $C_P[i]$ must be removed as P_1's support, and P_1 must be unassigned (Figure 29 lines 5-7). LTMS-WL's conservative resupport algorithm is the same as that of the LTMS; see section 3.2.2 for details.

4.1.2 LTMS-WL for Preprocessing

During preprocessing, where variable unassignments are not made in any predetermined sequence, LTMS-WL's unassign component will encounter situations where an unwatched literal is unassigned while some watched literal(s) in the same clause are FALSE.

For example, consider clauses:

$C_1 = (P_1 = \text{TRUE})$,

$C_2 = (\boldsymbol{\neg P_1 = \text{FALSE}} \vee \boldsymbol{P_2 = \text{TRUE}})$,

$C_3 = (\neg P_1 = \text{FALSE} \vee \boldsymbol{\neg P_2 = \text{FALSE}} \vee \boldsymbol{P_3 = \text{TRUE}})$,

$C_4 = (P_4 = \text{TRUE})$, and

$C_5 = (\neg P_1 = \text{FALSE} \vee \boldsymbol{\neg P_4 = \text{FALSE}} \vee \boldsymbol{P_5 = \text{TRUE}})$.

P_1 = TRUE, P_2 = TRUE, P_3 = TRUE, P_4 = TRUE, and P_5 = TRUE are supported by clauses C_1, C_2, C_3, C_4, and C_5 respectively. When C_1 is deleted, P_1 losses its support and is unassigned. At this point, neither C_2 nor C_3 could continue to support propositions P_2 and P_3. However, since literal $\neg P_1$ in C_3 is not watched, C_3 would not be visited if unassign only looked at clauses containing watched literals of P_1.

Therefore, during preprocessing, LTMS-WL's unassign algorithm must search through the clauses containing both watched and unwatched literals associated with an unassigned proposition P. However, if P's literal L evaluates to TRUE, then L's clause C could not support a proposition; furthermore, the watched literals in C do not need to be updated: if L is watched, then it could remain watched after the unassignment; if L is unwatched, then the watched literals could not be FALSE since L's value was TRUE. So the algorithm only needs to consider clauses where literals associated with P evaluate to FALSE. The use of watched literals in LTMS-WL's unassign algorithm for preprocessing reduces the number of clauses visited by a factor of ½ compared to adjacency lists.

```
unassign(P)
1    if P = TRUE
2         then C_P ← list of clauses with positive literals associated with P
3         else C_P ← list of clauses with negative literals associated with P
4    P ← UNKNOWN
5    for i ← 1 to length(C_P)
6         do if a watched literal, W, in C_P[i] is FALSE and
7               literal, L, associated with P is unwatched
8                  then watch L and unwatched W
9            if C_P[i] supports some proposition P_1
10                 then remove C_P[i] as P_1's support
11                       unassign(P_1)
12           if C_P[i] is a unit clause
13                 then insert(C_P[i], unitClauseList)
14   while not empty(unitClauseList)
15        do C_u←front(unitClauseList)
16           if C_u is a unit clause
17                 P←unassigned variable in C_u
18                 assign P so it's literal in C_u is TRUE
19                 propagate(P)
```
Figure 30: LTMS-WL Unassign Pseudo-code for Preprocessing

Figure 30 presents the pseudo-code for this algorithm. When a proposition P is unassigned, the algorithm visits the list of all clauses C_P containing either positive or negative literals of P (Figure 30 lines 1-3). If P's literal L in clause $C_P[i]$ is unwatched while some watched literal W is FALSE, then L replaces W as a watched literal in $C_P[i]$. If clause $C_P[i]$ supports some proposition P_1, then $C_P[i]$ must be removed as P_1's support, and P_1 must be unassigned (Figure 29 lines 5-7). LTMS-WL's conservative resupport algorithm is the same as that of the LTMS; see section 3.2.2 for details.

4.2 ITMS with watched literals

Recall from section 3.3, that the three main concepts behind the ITMS are propagation numbering, conflict repair during propagation, and aggressive resupport during variable unassignment. Among them, the propagation numbering system is simply a set of rules used to set the propagation number of a proposition after an assignment change. Applying these rules does not require search through any list of clauses, nor are these rules affected by watched literals within a supporting clause. Therefore, the propagation numbering system acts independently of the data structure used by the ITMS and remains unchanged in the ITMS-WL algorithm; see section 3.3.1 for details on propagation numbering.

Watched literals, however, could be applied to conflict repair and aggressive resupport to improve the performance of these algorithms. Algorithmic details are presented in sections 4.2.1 and 4.2.2 below.

4.2.1 Conflict Repair

Recall that during a context switch, the ITMS's propagate and conflict repair algorithms first propagates the newly added clauses before retracting dependents of the deleted clauses. Likewise, when conflict repair flips the assignment of a proposition P, the propagation algorithm first propagates P's new assignment V before retracting dependents of the old assignment $\neg$V. However, although designed to increase the chances of aggressive resupport, this propagate-before-unassign procedure could introduces clauses where some unwatched literals are unassigned while one or both of the watched literals are FALSE.

An alternative way to think about this is that watched literals do not need to be updated during variable unassignment if the most recently assigned propositions are the first retracted. However, because conflict repair propagates new assignments before retracting dependents of the old assignments, when the algorithm begins unassignment, dependents of the old assignment are no longer the most recently assigned variable. Therefore, watched literals must be updated during variable unassignment after conflict repair.

For example consider clauses:

$C_1 = (\ \mathbf{\neg P_1 = FALSE} \vee \mathbf{P_2 = TRUE}),$

$C_2 = (\neg P_2 = \text{FALSE} \vee \mathbf{\neg P_3} \vee \mathbf{P_4})$, and

$C_3 = (\mathbf{P_1 = TRUE} \vee \mathbf{P_3} \vee \neg P_5 = \text{FALSE}).$

P_1 = TRUE and P_5 = TRUE are supported by other clauses in the theory and are not dependent on each other; P_2 = TRUE is supported by C_1. Assume that P_1's value is flipped from TRUE to FALSE. If unassignment took place before propagation, then using the watched literals lists would be sufficient for unassign and propagate. The algorithm will first unassign P_1. C_1 is then identified through watched literal $\neg P_1$. Since P_1 is unassigned, C_2 can no longer support P_2; thus P_2 is unassigned as well, and the clauses become:

$C_1 = (\ \mathbf{\neg P_1} \vee \mathbf{P_2}\ ),$

$C_2 = (\neg P_2 \vee \mathbf{\neg P_3} \vee \mathbf{P_4})$, and

$C_3 = (\mathbf{P_1} \vee \mathbf{P_3} \vee \neg P_5 = \text{FALSE}).$

Next, the algorithm propagates the assignment P_1 = FALSE. C_1 is not visited because watched literal $\neg P_1$ evaluates to TRUE. C_3, however, is identified as a unit clause through FALSE watched literal P_1, and P_3 is assigned to TRUE with C_3 as its support. C_2 is then identified through watched literal $\neg P_3$. Since $\neg P_3$ equals FALSE while unwatched literal $\neg P_2$ is unassigned, $\neg P_2$ replaces $\neg P_3$ as a watched literal in C_2, and the clauses become:

$C_1 = (\neg\mathbf{P_1} = \mathbf{TRUE} \lor \mathbf{P_2})$,

$C_2 = (\neg\mathbf{P_2} \lor \neg P_3 = \text{FALSE} \lor \mathbf{P_4})$, and

$C_3 = (\mathbf{P_1} = \mathbf{FALSE} \lor \mathbf{P_3} = \mathbf{TRUE} \lor \neg P_5 = \text{FALSE})$.

However, if propagate takes place before unassign, then simply using the list of watched literals would be insufficient to maintain the integrity of the watched-literals data structure. For the same example above,

$C_1 = (\neg\mathbf{P_1} = \mathbf{FALSE} \lor \mathbf{P_2} = \mathbf{TRUE})$,

$C_2 = (\neg P_2 = \text{FALSE} \lor \neg\mathbf{P_3} \lor \mathbf{P_4})$, and

$C_3 = (\mathbf{P_1} = \mathbf{TRUE} \lor \mathbf{P_3} \lor \neg P_5 = \text{FALSE})$.

P_1 = TRUE and P_5 = TRUE are supported by other clauses in the theory and are not dependent on each other; P_2 = TRUE is supported by C_1. Also assume that P_1's value is flipped from TRUE to FALSE. When propagating P_1 = FALSE, C_3 is identified as a unit clause through FALSE watched literal P_1, and P_3 is assigned to TRUE with C_3 as its support. C_2 is then identified as a unit clause through watched literal $\neg P_3$, since P_2 was not unassigned. So P_4 is assigned to TRUE with C_2 as its support and the clauses becomes:

$C_1 = (\neg\mathbf{P_1} = \mathbf{TRUE} \lor \mathbf{P_2} = \mathbf{TRUE})$,

$C_2 = (\neg P_2 = \text{FALSE} \lor \neg\mathbf{P_3} = \mathbf{FALSE} \lor \mathbf{P_4} = \mathbf{TRUE})$, and

$C_3 = (\mathbf{P_1} = \mathbf{FALSE} \lor \mathbf{P_3} = \mathbf{TRUE} \lor \neg P_5 = \text{FALSE})$.

Next, when unassigning dependents of the assignment P_1 = TRUE, C_1 will be identified through watched literal $\neg P_1$; since $\neg P_1$ = TRUE, C_1 can no longer support P_2 = TRUE, so P_2 is unassigned. However, the algorithm cannot identify clause C_2 through unwatched literal P_2 if only clauses containing watched literals of P_2 are looked at. If this happens then C_2 becomes $(\neg P_2 \lor \neg\mathbf{P_3} = \mathbf{FALSE} \lor \mathbf{P_3} = \mathbf{TRUE})$, where watched literal $\neg P_3$ is FALSE while unwatched literal $\neg P_2$ is unassigned. Therefore, when retracting variable assignments after propagation, the algorithm must search through the list of all FALSE literals associated with an unassigned proposition, not just the watched literals

```
propagate(P)
1     if P = TRUE
2         then C_P^T ← list of clauses with positive literals associated with P
3              C_P^F ← list of clauses with negative watched literals associated with P
4         else C_P^T ← list of clauses with negative literals associated with P
5              C_P^F ← list of clauses with positive watched literals associated with P
6     noConflict ← true
7     for i ← 1 to length(C_P^F)
8         do W^1 ← watched literal in C_P^F[i] associated with P
9            W^2 ← other watched literal in C_P^F[i]
10           replace W^1 with unwatched literal if possible
11           if W^1 cannot be replaced and W^2 = UNKNOWN
12               then P_1 ← proposition associated with W^2
13                    if W^2 is POSITIVE
14                       then P_1 ← TRUE
15                       else P_1 ← FALSE
16                    record C_P^F[i] as the support for P_1
17                    N_P1 ← 1+ max(propagation # of other propositions in clause)
18                    if propagate(P_1) = false
19                       then noConflict ← false
20           if W^1 cannot be replaced and W^2 = FALSE
21               then if not repairConflict(C_P^F[i])
22                       then noConflict ← false
23    if (P has been flipped)
24        then for j ← 1 to length(C_P^T)
25                 do if P's literal L is not watched and
26                       a watched literal W is FALSE
27                       then change the watch from W to L
28                    if C_P^T[j] supports a proposition
29                       then P_2 ← proposition supported by C_P^T[j]
30                            unassign(P_2)
31    return noConflict
```

Figure 31: ITMS-WL Propagation Pseudo-code

Figure 31 and Figure 32 contains the propagation and conflict repair algorithms for ITMS-WL. These algorithms are the same as Figure 20 and Figure 21 in section 3.3.2 except for the watched-literals specific details. During the forward assignment phase of a proposition P (Figure 31 lines 7-22), only clauses, C_P^F, containing FALSE watched literals associated with P require updates (Figure 31 lines 1, 3, 5). W^1 is the watched literal in clause $C_P^F[i]$

that is associated with P, and W^2 is the other watched literal. Since W^1 evaluates to FALSE, the algorithm attempts to replace W^1 with a non-false, unwatched literal if possible (Figure 31 line 10). If W^1 cannot be replaced, then $C_P^F[i]$ is unit if W^2 is unassigned and violated if W^2 = FALSE (Figure 31 lines 11, 20); if W^2 = TRUE, then $C_P^F[i]$ is satisfied, so no propagation is necessary. If $C_P^F[i]$ is unit, then the proposition P_1 associated with W^2 is assigned such that W^2 evaluates to TRUE (Figure 28 lines 11-13). And, like in section 3.2.2, $C_P^F[i]$ is recorded as P_1's support. If $C_P^F[i]$ is violated, then the conflict repair algorithm is called upon to repair $C_P^F[i]$ if possible.

If P was not formerly unassigned then the algorithm must also unassign dependents of P's old assignment (Figure 31 lines 7-22). During this unassignment phase within propagate, all clauses, C_P^T, containing TRUE literals of P (watched and unwatched) must be updated (Figure 31 lines 1, 2, 4). If P's literal L in clause $C_P^T[i]$ is unwatched while one of the watched literals, W, is FALSE, then L replaces W as a watched literal in $C_P^T[i]$. Also, any proposition P_2 supported by clause $C_P^T[i]$ must be unassigned (Figure 31 lines 23-27).

```
repairConflict(C)
1       P ← Proposition in C with largest propagation number
2       if P has not been flipped before
3           then if P = TRUE
4               then P ← FALSE
5               else P ← TRUE
6             if literal of P not watched
7               then replace a watched literal with P's literal
8             N_P ← 1+ max(propagation # of other propositions C)
9             record C as the support for P
10            if (propagate(P) = false)
11              then return false
12            return true
13          else return false
```

Figure 32: ITMS-WL Conflict Repair Pseudo-code

The conflict repair algorithm for the ITMS-WL is the same as that for the ITMS (see section 3.3.2) except for lines 6-7 where an unwatched literal, L, of proposition P replaces a watched literal in clause C. Recall, a clause C is violated if all of its literals, including the watched literals, are FALSE. After C is repaired by flipping the value of proposition P, L becomes TRUE. Thus, if L was not watched, then it should become watched in place of one of the FALSE literals.

When propagate is used to assign an unassigned proposition P, the algorithm simply searches through the list of clauses containing FALSE watched literals associated with P; since P was unassigned, there are no assignments dependent on the former value of P, so clauses containing TRUE literals of P do not need to be searched over. Therefore, the number of clauses visited by ITMS-WL's propagation algorithm is 1/avg-#-of-literals-per-clause less than an ITMS using the counter-based method. However, if propagate is used to flip the truth assignment of P, then the algorithm must search through clauses containing FALSE watched literals and all TRUE literals of P. In these situations, ITMS-WL's propagation algorithm only has a saving of ½ *(1+ 1/avg-#-of-literals-per-clause) over number of clauses visited by an ITMS using the counter-based method.

4.2.2 Aggressive Resupport

Section 4.2.1 showed that if unassignment of a proposition P is triggered by conflict repair, then the algorithm must search through all clauses associated with FALSE literals of P. The same is true if unassign is used after a context switch where newly added clauses are propagated before dependents of the deleted clauses are retracted.

For example, consider clause $C_1 = (\neg P_1 = \text{FALSE} \vee \mathbf{\neg P_2} \vee \mathbf{P_3})$, where P_1 = TRUE is supported by some other clause in the theory. Assume that P_1 loses its support while clause $C_2 = (P_2)$ added. If propagate takes place before unassign, then simply using the list of watched literals would be insufficient to maintain the integrity of the watched-literals data structure. The algorithm will assign P_2 to TRUE with C_2 as its support. Next C_1 is identified as a unit clause through watched literal $\neg P_2$ and is propagated to support P_3 = TRUE; propagation terminates. However, when P_1 is unassigned, C_1 would not be identified through unwatched literal $\neg P_1$ if the algorithm simply searches through the list of watched literals associated with C_1. If this happens then C_1 becomes $(\neg P_1 \vee \mathbf{\neg P_2} =$

FALSE $\vee$ **P_3 = TRUE**), where watched literal $\neg P_2$ is FALSE while unwatched literal $\neg P_1$ is unassigned. Therefore, when retracting variable assignments after propagation, the unassign algorithm must always search through the list of all FALSE literals associated with an unassigned proposition.

Figure 33 presents the pseudo-code of the ITMS-WL's unassign algorithm. It is similar to the LTMS-WL's unassign algorithm for preprocessing (see Figure 30) but uses aggressive resupport instead of conservative resupport (Figure 33 line 12). When a proposition P needs to be unassigned, the algorithm must look through clauses, C_P, containing all FALSE literals of P (Figure 33 lines 1-3). For each clause $C_P[i]$, if its literal, L, associated with P is not watched while a watched literal, W, is FALSE, then L replaces W as a watched literal in $C_P[i]$ (Figure 33 lines 7-9). Also, if $C_P[i]$ supports some proposition P_1, then ITMS-WL first attempts to resupport P_1, and only unassigns P_1 if a resupport could not be found (Figure 33 lines 10-13).

```
unassign(P)
1      if P = TRUE
2             then C_P ← list of clauses with positive literals associated with P
3             else C_P ← list of clauses with negative literals associated with P
4      P ← UNKNOWN
5      N_P ← 0
6      for i ← 1 to length(C_P)
7             do if a watched literal, W, in C_P[i] is FALSE and
8                  literal, L, associated with P not watched
9                    then watch L and unwatched W
10                if C_P[i] supports some proposition P_1
11                   then remove C_P[i] as P_1's support
12                        if not resupport(P_1)
13                           then unassign(P_1)
```
Figure 33: ITMS-WL Unassign Pseudo-code

The resupport algorithm for ITMS-WL benefits greatly from the use of watched literals. When searching for a resupport for proposition P, a clause C is suitable only if its literal, L, associated with P evaluates to TRUE while all other literals evaluates to FALSE. This means that L must be one of the watched literals in C. Also, if at least one of the watched literals in C is FALSE then all of the unwatched literals must also be FALSE. Thus, when looking to resupport P, the algorithm only needs to consider the list of clauses containing TRUE watched literals associated with P. For each clause containing such a literal, the clause can provide resupport for P if its other watched literal evaluates to FALSE, and the propagation numbers of all other literals are less than N_P. Figure 34 presents the pseudo-code for this ITMS-WL aggressive resupport algorithm

```
resupport(P)
    if P = TRUE
        then C_P ← list of clauses with positive watched literals associated with P
        else C_P ← list of clauses with negative watched literals associated with P
    for j ← 1 to length(C_P)
        do if other watched literals in C_P[j] is FALSE and
              N_P > propagation number of all other propositions in clause
                then set C_P as P's new support
                    return true
    return false
```

Figure 34: ITMS-WL Aggressive Resupport Pseudo-code

During unassign, a counter-base approach must update the counters of clauses containing both TRUE and FALSE literals of unassigned proposition P, while ITMS-WL only considers clauses contain FALSE literals of P. Therefore, ITMS-WL's unassign algorithm on average searches over only ½ as many clauses as a counter-based ITMS. The ITMS-WL resupport algorithm, on the other hand, has a saving of 2/avg-#-of-literals-per-clause over the number of clauses visited by an ITMS using adjacency lists.

4.3 Summary

Chapter 4 detailed the LTMS-WL and the ITMS-WL. These algorithms retained the LTMS and ITMS's ability to perform incremental assignment changes to propositions while reducing the number of clauses visited through the use of the watched-literals data structure. The exact savings achieved by the combined algorithms vary from component to component, but the use of watched literals never adversely affect the number of propositional assignments retained or the number of clauses visited by these algorithms

For the remainder of this thesis, we first present two SAT solvers, ISAT and zCHAFF, that we applied the LTMS-WL and the ITMS-WL algorithms to. Then empirical performance results of these incremental unit propagation algorithms with watched literals are presented and compared against their counter-based and non-incremental counterparts.

Chapter 5

SAT Solvers with Incremental Unit Propagation

Two SAT solvers are used for the empirical evaluation of the logic-based truth maintenance system with watched literals (LTMS-WL) and the incremental truth maintenance system with watched literals (ITMS-WL) algorithms: ISAT and zCHAFF. Both are DPLL based solvers that follow the upper level pseudo-code found in Chapter 2 Figure 1. ISAT is used to compare LTMS-WL and ITMS-WL's watched-literals data structure against their counter-based counterparts; and zCHAFF is used to compare the incremental algorithms, LTMS-WL and ITMS-WL, against a non-incremental unit propagation algorithm using watched-literals and stack-based backtracking. The details of these solvers are presented in sections 5.1 and 5.2.

5.1 ISAT

ISAT is a simple incremental SAT solver that uses truth maintenance within the basic DPLL algorithm. We have developed four variants of ISAT each using a different TMS algorithm for unit propagation and assignment retraction during the preprocessing, decision, and backtracking components of the SAT solver. ISAT-LTMS-C and ISAT-ITMS-C use a counter-based data structure with the LTMS and the ITMS algorithms found in sections 3.2.2 and 3.3. ISAT-LTMS-WL and ISAT-ITMS-WL, on the other hand, use the LTMS-WL and ITMS-WL algorithms described in Chapter 4. Other components of ISAT remain the same across the different versions and work in the same way as the algorithms used by the simple SAT example in Chapter 2.

5.1.1 Preprocessing

ISAT's preprocessing component allows for the addition of a new SAT theory or addition and deletion of clauses from an existing theory. Since the addition of a new theory does not include deleted clauses, incremental unit propagation is not necessary; therefore, preprocessing performs a simple unit propagation step using either counters (see section 3.1.1) or watched literals (see section 3.1.3). After addition and deletion of clauses, preprocessing performs incremental propagation and retraction of variable assignments using one of counter-based LTMS, counter-based ITMS, LTMS-WL, or ITMS-WL. Preprocessing returns UNSATISFIABLE if the initial propagation step encounters a violated clause, SATISFIABLE if all propositions are assigned without violating any clauses, or UNDETERMINED otherwise.

5.1.2 Decision and Conflict Analysis

If satisfiability of the theory cannot be determined during preprocessing, ISAT moves on to the search process. During search, decision variables are selected from the list of unassigned propositions with no special preferences. Decision variables are always assigned to TRUE before FALSE. And after an assignment, the deduction component is invoked.

If a violated clause is encountered during search, conflict analysis searches through the list of decision variables starting with the one most recently assigned. If the variable is FALSE, the algorithm unassigns it and moves on to the next most recently assigned decision variable; the process repeats until a TRUE decision variable is encountered. Then, that variable's assignment is flipped to FALSE and deduction and backtracking is invoked. If a TRUE decision variable cannot be found, then the theory is UNSATIAFIABLE.

5.1.3 Deduction and Backtracking

The deduction and backtracking components use unit propagation and unassignment in the same way as preprocessing. If deduction is called after a decision variable assignment with no search backtracking, then a simple unit propagation step is performed using either counters or watched literals. However, if called after conflict analysis, the algorithm propagates the decision variable assignment and retracts the unassignment(s) using one of counter-based LTMS, counter-based ITMS, LTMS-WL, or ITMS-WL. If a violated clause is encountered during deduction, the conflict analysis is invoked. Else if all unit clauses are propagated without encountering a violated clause, then the search continues with another decision step. The theory is SATISFIABLE if all propositions are assigned without violating any clauses.

5.2 zCHAFF

zCHAFF 2004.11.15 [20, 7] is a state-of-the-art SAT solver built by Princeton's Boolean Satisfiability Research Group. It uses the watched-literals data structure with stack-based backtracking (see sections 3.1.3 and 3.2.1} as its unit propagation algorithm within the preprocessing, deduction, and backtracking components. zCHAFF also incorporates into the basic DPLL structure other cutting edge SAT techniques including the Variable State Independent Decaying Sum Decision Heuristic (VSIDS) [20, 31] and 1stUIP Shirking conflict analysis [30, 31, 21].

5.2.1 Preprocessing

zCHAFF preprocessing allows clauses to be added and deleted in groups. Typically, all original clauses in the initial theory belong to group 0, and the next set of incrementally added clauses is assigned to group 1 and so on. Clause deletion is achieved by providing the solver with a clause group ID number, and all clauses within that group will be deleted.

Since the unit propagation algorithm within zCHAFF uses a stack-based backtracking scheme, when a group of clauses is deleted from the theory, all propositions in the theory must be unassigned. (The original zCHAFF solver will hereon be referred to as zCHAFF-stack.) We also created 2 modified zCHAFF solvers referred to as zCHAFF-LTMS and zCHAFF-ITMS. These solvers incorporate into the preprocessing component of zCHAFF the LTMS-WL and ITMS-WL algorithms, and perform variable assignments and unassignments incrementally.

5.2.2 Decision

VSIDS keeps 2 counters with each proposition containing the number of positive and negative literals of that proposition. During a decision step, the algorithm simply selects an unassigned proposition and polarity with the highest counter value and assigns the proposition such that the chosen polarity evaluates to TRUE. Ties are broken randomly, and periodically, all counters are divided by a constant factor (1/2 in zCHAFF).

Higher counter values corresponds to a larger number of clauses satisfied by the assignment; and the periodic reduction of the counter values places higher emphasis on more recently added clauses. Within search, clauses can be added by the conflict analysis algorithm described below. For more information on VSIDS, see [20, 31].

5.2.3 Conflict Analysis

When a clause C is violated, 1[st]UIP conflict analysis traces the cause of that violation through supports of C's literals. For example, if $C = (\neg P_1 = FALSE \vee P_2 = FALSE)$, and $C_1 = (P_1 = TRUE \vee P_3 = FALSE)$ is the support of P_1, then C and C_1 is merged with P_1

removed to create the new clause C_2 = (P_2 = FALSE $\vee$ P_3 = FALSE). Since C_2 is also violated, the same process is repeated until clause C_i is found such that literal L in C_i has the single largest decision level (DL) in C_i (no ties), and L's proposition is a decision variable. Thus, when C_i is added to the clause databases and the search backtracks to DL, C_i becomes a unit clause with only L unassigned and can therefore be propagated so that L evaluates to TRUE. The addition of C_i simultaneously prunes the search space leading to C_i's violation, and brings the search to a new search space by flipping the value of L.

However, if the number of literals in C_i exceeds a certain limit, then the shrinking technique is employed to shorten the length of future conflict clauses. Shrinking orders the literals in C_i by their decision levels, backtracks to the highest decision level in C_i, and reassigns C_i's literals to FALSE until a violated clause is encountered. This process generally decreases the number of assigned variables and compacts the supporting clauses derived. For more details on 1stUIP conflict analysis with shrinking, see [30, 31, 21].

5.2.4 Deduction and Backtracking

zCHAFF's deduction and backtracking components uses unit propagation with watched-literals and stack-based backtracking. And these components are unaltered for zCHAFF-LTMS and zCHAFF-ITMS for the following reasons.

1. The VSIDS selects the new decision variables among the unassigned propositions. Since an ITMS conserves assignments across context switches, it may alter the variables selected by VSIDS.
2. The conflict analysis algorithm is very dependent on supporting clauses for propositional assignments which may change with incremental variable assignment and resupport.

Since, it is unclear if and how incorporating an TMS within the search may affect zCHAFF's performance and vise versa, the decision and backtracking components of zCHAFF are left unchanged.

Chapter 6

Results and Analysis

Chapter 6 contains the empirical results of the LTMS-WL and ITMS-WL algorithms evaluated using the 4 ISAT and 3 zCHAFF solvers described in Chapter 5. Recall, TMS algorithms improve performance gain by decreasing the number of unnecessary variable assignment changes; and the watched-literals conserves computational effort by decreasing the number of clauses visited during unit propagation. Thus, we use the following parameters to evaluate performance of the TMS with watched-literals algorithms: the number of variable assignments (PA), unassignments (PUA), and resupports (PR) and the number of clauses visited after assignment (CA), unassignment (CUA), and resupport (CR). Also, in order to test incremental unit propagation performance using real world problems, we interfaced these SAT solvers to a Mode Estimation program, and tested their performances through a series of estimation steps using models of space systems. Section 6.1 briefly describes mode estimation and the models that we used for testing. And section 6.2 and 6.3 present the performances of the ISAT and zCHAFF solvers.

6.1 Evaluation Setup

Mode Estimation [17, 18] monitors and diagnoses robotic system behavior using declarative models of the system called Probabilistic Concurrent Constraint Automata (PCCA). A mode estimator determines the most likely states that a system is in by reasoning over the PCCA model of that system along with commands and sensory observations. A SAT solver is used within the estimator to determine whether a certain set of states is consistent with the given model, commands, and observations. As the estimator searches over different possible states (called belief states), the SAT theory is incrementally updated to reflect the different state assignments while the model information remains constant.

Four PCCA models are used for testing:

1. The EO1 model models the Hyperion Imager, Advanced Land Imager, WARP data recording device, and other data transferring components launched aboard the Earth Observing One satellite launched on November of 2000 [9].
2. The Mars EDL model contains critical propulsion and navigational components required for a Mars entry, decent, and landing sequence [11].
3. The SPHERES model models the propulsion subsystem of the Synchronized Position Hold, Engage, and Reorient Experimental Satellites (SPHERES) developed by MIT's Space Systems Laboratory and Payload Systems, Inc. [23].
4. The ST7 model contains the communication subsystem of the Space Technology 7 concept study of NASA's New Millennium program [5].

These models use variables of non-binary domains, and must be converted into SAT theories before they can be used by the SAT solvers. Table 1 contains the models' properties after they are converted to binary CNF format.

Table 1: PCCA Model Properties

Model	# Propositions	# Clauses	# Literals
EO1	233	725	1627
Mars EDL	141	646	1435
Spheres	242	4610	36012
ST7	90	353	782

Testing is performed by automatically generating a series of estimation steps for each of these models. When an estimation step calculates the likelihood of future states, one or more context switches are made to the SAT theory.

6.2 ISAT Performance Results

Recall that 4 different versions of ISAT were implemented each containing one of LTMS-C, LTMS-WL, ITMS-C, and ITMS-WL algorithms. Table 2 contains the performance results of these ISAT solvers using the EO1, Mars EDL, SPHERES, and ST7 models described above. We ran a total of 1048 context switches using the EO1 model, 257 contest switches using the Mars EDL model, 100 context switches using the SPHERES model each, and 100 context switches using the ST7 model.

Table 2: ISAT Data

Solver	Model	PA	PUA	PR	CA	CUA	CR
ISAT-LTMS-C	EO1	62149	61916	0	549836	724764	0
	Mars EDL	3943	3802	0	35294	45000	0
	SPHERES	24200	23958	0	2485400	4165946	0
	ST7	9000	8910	0	78200	103803	0
	Total	**99,292**	**98,586**	**0**	**3,148,730**	**5,039,513**	**0**
ISAT-LTMS-WL	EO1	63270	63037	0	123839	179528	0
	Mars EDL	4041	3900	0	8924	11413	0
	SPHERES	34600	34358	0	790483	1738018	0
	ST7	12248	12158	0	37004	35483	0
	Total	**114,159**	**113,453**	**0**	**960,250**	**1,964,442**	**0**
ISAT-ITMS-C	EO1	3140	2702	5125	19794	16422	15362
	Mars EDL	470	250	4054	4854	2538	5250
	SPHERES	28834	24531	3208	1780723	1012944	567290
	ST7	9488	7961	1770	78249	63781	35671
	Total	**41,932**	**35,444**	**14,157**	**1,883,620**	**1,095,685**	**623,573**
ISAT-ITMS-WL	EO1	3164	2724	5128	7257	5705	9161
	Mars EDL	472	251	4053	2172	1130	4344
	SPHERES	28314	23947	2851	1277983	507781	356967
	ST7	9473	7956	1690	37477	22443	9244
	Total	**41,423**	**34,878**	**13,722**	**1,324,889**	**537,059**	**379,716**

PA = number of propositions assigned.
PUA = number of propositions unassigned.
PR = number of propositions aggressively resupported.
CA = number of clauses visited by propagation after an variable assignment.
CUA = number of clauses visited by unassign after an variable unassignment.
CR = number of clauses visited by resupport in order to resupport a propositional assignment.

Figure 35 plots the total number of propositions assigned, unassigned, and resupported for each of the 4 ISAT solvers. The x-axis is divided into 3 groups: propositions assigned, propositions unassigned, and propositions resupported. And the y-axis plots the total number of assignments, unassignements, and resupports of the 4 models combined. First observe that the number of propositions assigned, unassigned, and resupported are similar for ISAT-LTMS-C and ISAT-LTMS-WL and for ISAT-ITMS-C and ISAT-ITMS-WL, but are quite different between the LTMS and ITMS algorithms. This is expected because assignment changes and resupports should not be affected by the data structure used. (The resupport values for ISAT-LTMS-C and ISAT-LTMS-WL are zero because an LTMS does not use aggressive resupport.)

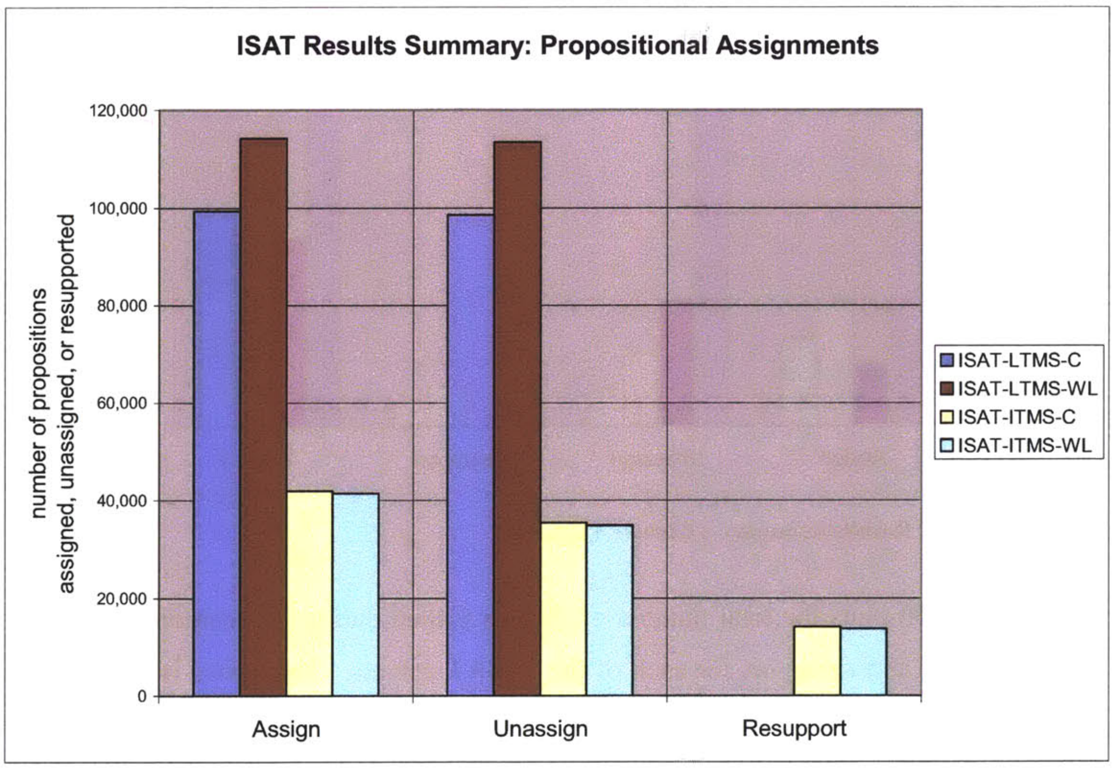

Figure 35: ISAT Results Summary – Propositional Assignments

However, there are some variations within these propositional assignment variables, particularly between those of the LTMS-C and the LTMS-WL, which is also not surprising given the decision algorithm used by ISAT. Recall that the decision algorithm selects a decision variable from available propositions. However, due to the difference in the variable ordering within counter-based adjacency lists and watched-literals lists, propositions may not be unassigned in the same order. Thus for any particular decision step, ISAT-LTMS-C and ISAT-LTMS-WL may select a different proposition as the new decision variable, which could lead to a difference in exact number of propositions assigned, unassigned, and resupport.

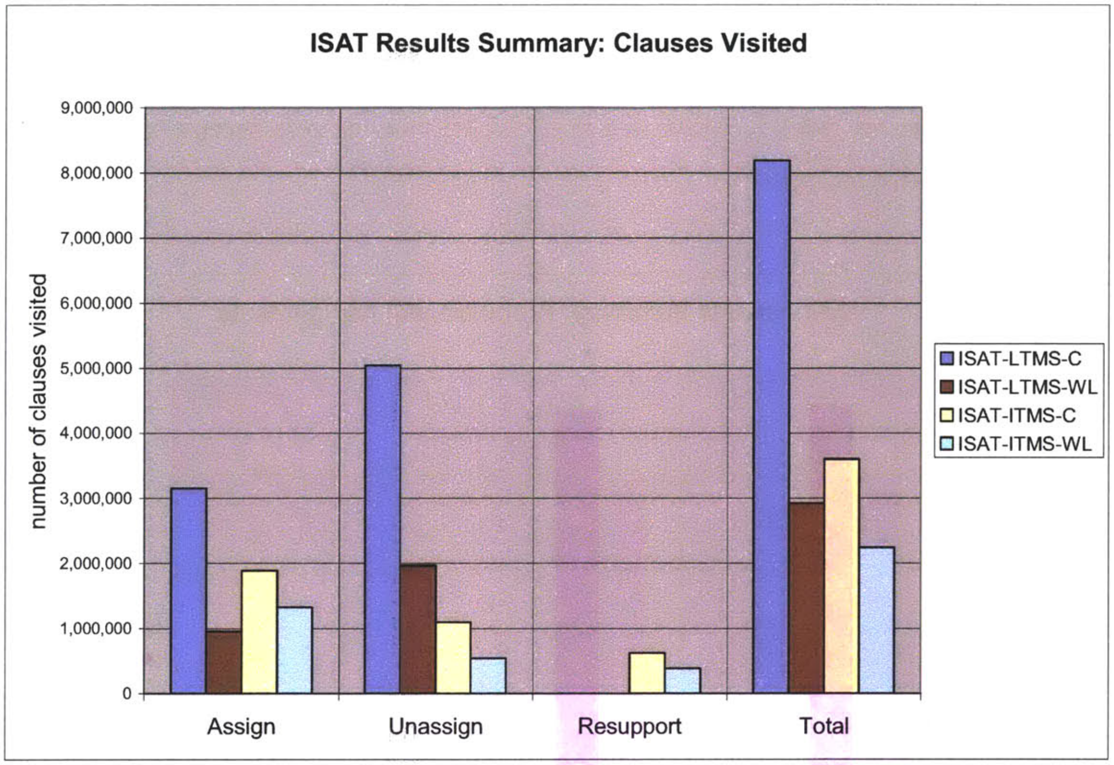

Figure 36: ISAT Results Summary – Clauses Visited

Next, Figure 36 plots the total number of clauses visited during propositional assignment, unassignment, and resupport for each of the 4 ISAT solvers. The x-axis is divided into 4 groups: clauses visited during assign, clauses visited during unassign, clauses visited during resupport, and clause visited during assign, unassign, and resupport. And the y-axis plots the total number of clauses visited by the 4 models combined. Note that the number of clauses visited by LTMS and ITMS dramatically decreases with the use of watched literals. For these test cases, the number of clauses visited by the LTMS-WL algorithm during unit propagation is 64% less then the number of clauses visited by ITMS-C. There is also a 37% saving in the number of clauses visited by the ITMS-WL algorithm compared to the ITMS-C.

6.3 zCHAFF Performance Results

Recall that 3 zCHAFF solvers are used to compare the performance of the incremental TMS algorithms against the non-incremental stack-based algorithm. For each of these solvers, we ran a total of 71 context switches using the EO1 model, 213 contest switches using the Mars EDL model, 100 context switches using the SPHERES, and 100 context switches using the ST7 model. Table 3 contains the performance data for these solvers during the preprocessing step. Since the algorithm used for deduction and backtracking are the same between the 3 versions of zCHAFF, performance data during search is not analyzed.

Table 3: zCHAFF Preprocessing Data

Solver	Model	PA	PUA	PR	CA	CUA	CR
	EO1	3964	3858	0	6599	0	0
zCHAFF-stack	Mars EDL	16783	16679	0	48478	0	0
	SPHERES	13800	13662	0	733039	0	0
	ST7	5752	5699	0	17676	0	0
	Total	**40,299**	**39,898**	**0**	**805,792**	**0**	**0**
	EO1	224	118	0	505	346	0
zCHAFF-LTMS	Mars EDL	7021	6917	0	24111	26590	0
	SPHERES	11204	11066	0	726337	1953180	0
	ST7	4397	4344	0	15213	16370	0
	Total	**22,846**	**22,445**	**0**	**766,166**	**1,996,486**	**0**
	EO1	1589	1003	1189	7615	2908	10041
zCHAFF-ITMS	Mars EDL	5871	3433	3222	38778	13358	26454
	SPHERES	8519	6527	3440	708921	1656500	594126
	ST7	3222	2061	1218	18143	7778	12735
	Total	**19,201**	**13,024**	**9,069**	**773,457**	**1,680,544**	**643,356**

PA = number of propositions assigned.
PUA = number of propositions unassigned.
PR = number of propositions resupported.
CA = number of clauses visited by propagation after an variable assignment.
CUA = number of clauses visited by unassign after an variable unassignment.
CR = number of clauses visited by resupport in order to resupport a propositional assignment.

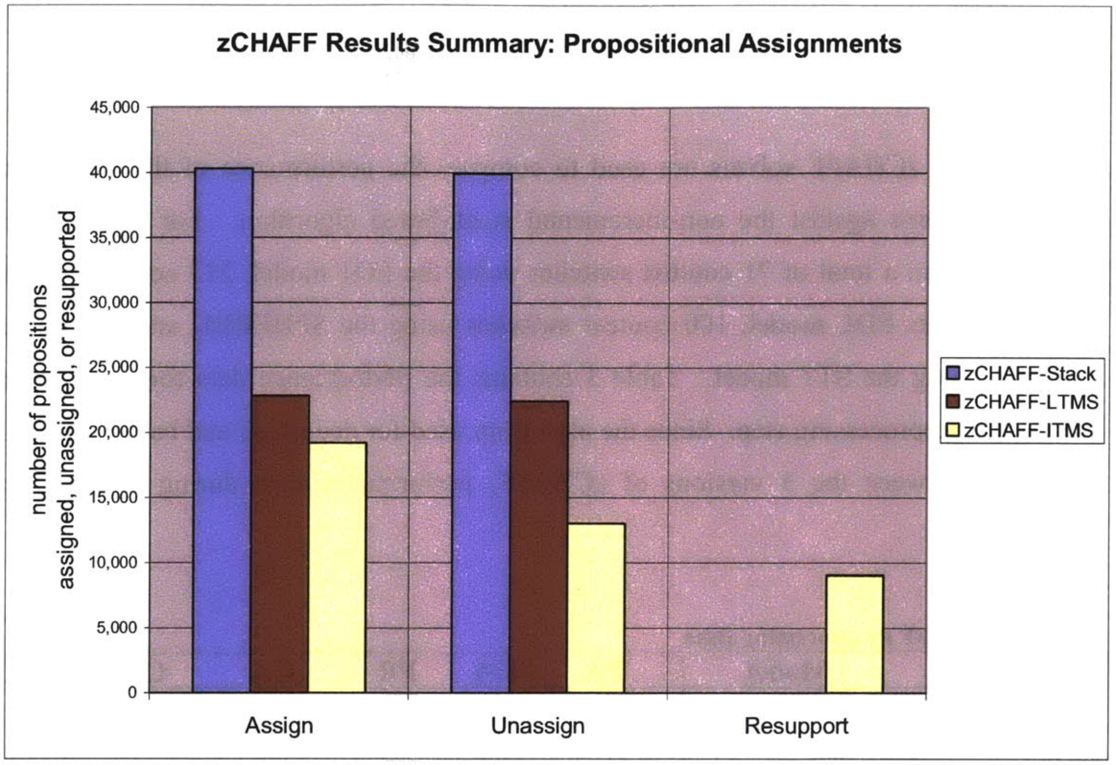

Figure 37: zCHAFF Results Summary – Propositional Assignments

Figure 37 plots the total number of propositions assigned, unassigned, and resupported for each of the 3 zCHAFF solvers; and Figure 38 plots the total number of clauses visited during propositional assignment, unassignment, and resupport. Note, the zCHAFF-LTMS solver saves 43% in both the number of propositional assignments and unassignments over the zCHAFF-stack. And zCHAFF-ITMS provides additional reductions in the number of propositional assignments and unassignments by 16% and 42% over zCHAFF-LTMS. However, since zCHAFF-stack is non-incremental, no clauses are visited during unassignment. zCHAFF-LTMS and zCHAFF-ITMS, on the other hand, must search through a considerable number of clauses in order to incrementally unassign dependents of the deleted clauses. The number of clauses searched by zCHAFF-LTMS and zCHAFF-ITMS during backtracking and resupport are around 73% of the total number of clauses visited by these algorithms.

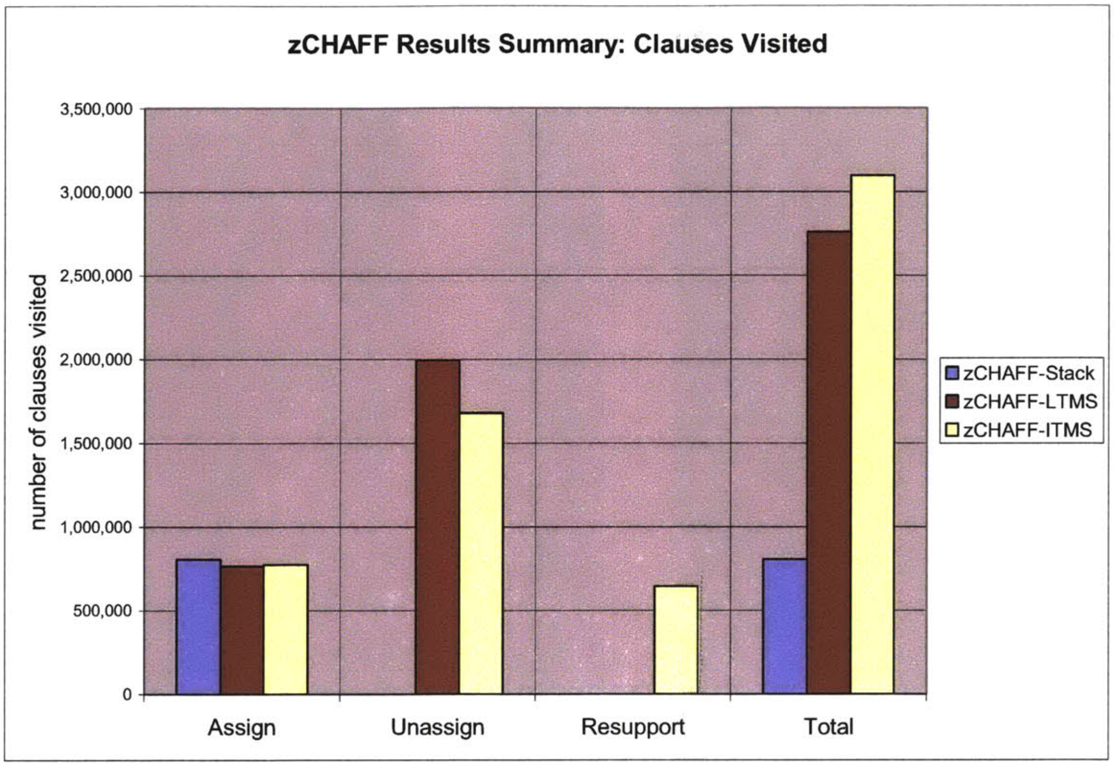

Figure 38: zCHAFF Results Summary – Clauses Visited

Chapter 7

Conclusion and Future Work

Two novel incremental unit propagation algorithms are developed within this thesis: Logic-based Truth Maintenance System with watched-literals and Incremental Truth Maintenance System with watched-literals. The LTMS-WL and ITMS-WL incorporate the watched-literals data structure into the LTMS and the ITMS, respectively. Empirical results show that these new algorithms decrease the number of clauses visited without affecting the incremental property of the LTMS and the ITMS. However, when compared to non-incremental unit propagation with stack-based backtracking, there is a performance tradeoff where decreasing the number of propositional assignments through these incremental algorithms increases the number of clauses visited by the SAT solver. Furthermore, incremental unit propagation was not applied to the search component of zCHAFF due to the presence of other, potentially conflicting, components of the solver.

For future work, we would like to determine if and how the LTMS-WL and the ITMS-WL affect the performance of various decision and conflict analysis algorithms, and vise versa. We have also conceived, but were unable to complete, an alternative incremental unit propagation algorithm called decision-level (DL) ITMS that could be fitted to the frame work of stack-based backtrack search. DL-ITMS is built upon the concept that the decision levels within tree search can be used to replace the propagation numbering system within the ITMS. Since decision levels are assigned and unassigned in order, a proposition P assigned at decision level DL can find well-founded supported in any clause C when the decision levels for C's other literals are less than DL. This ensures that P is assigned after the other literals, and thus a loop support could not be formed if C supports P. Also, the decision level of a proposition is already available within tree search and thus can be kept and used at no extra cost to the algorithm. During unassignment, variables can still be backtracked off of the assignment stack. However, an aggressive resupport strategy can be employed such that resupported propositions are moved to a new assignment stack, while those propositions that could not be resupported are unassigned.

Bibliography

[1] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, 1971.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2001.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. In *Communications of the ACM*, 5: 394-397, 1962

[4] J. Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3): 231-272, 1979.

[5] L. Fesq, M. Ingham, M. Pekala, J. Van Eepoel, D. Watson, and B. Williams. Model-based autonomy for the next generation of robotic spacecraft. In *Proceedings of the 53rd International Astronautical Congress*, 2002.

[6] K. Forbus and J. Kleer. *Building Problem Solvers*. The MIT Press, Cambridge, MA, 1993.

[7] Z. Fu, Y. Mahajan, and S. Malik. SAT competition – solver description: new features for the SAT'04 version of zChaff. In *the 7th International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[8] I. Gent and T. Walsh. The search for satisfaction. Internal Report, Department of Computer Science, University of Strathclyde, 1999.

[9] S. Hayden, A. Sweet, and S. Christa. Livingstone model-based diagnosis of Earth Observing One. In *Proceedings of the AIAA Intelligent Systems*, 2004.

[10] J. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 1995.

[11] M. Ingham. Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences. PhD thesis, MIT, 2003.

[12] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[13] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, 1992.

[14] I. Lynce and J. Marques-Silva. Efficient data structures for backtrack search SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 43: 137-152, 2005.

[15] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence*, 1999.

[16] J. Marques-Silva and K. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48: 506-521, 1999.

[17] O. Martin. Accurate belief state update for probabilistic constraint automata. Master's thesis, MIT, 2005.

[18] O. Martin, B. Williams, and M. Ingham. Diagnosis as approximate belief state enumeration for probabilistic concurrent constraint automata. In *Proceedings of the 20th National Conference on Artificial Intelligence*, 2005.

[19] D. McAllester. Truth maintenance. In *Proceedings of the 8th National Conference on Artificial Intelligence*, 1990.

[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. CHAFF: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, 2001.

[21] A. Nadel. Backtrack search algorithms for propositional logic satisfiability: review and innovations. Master's Thesis, Hebrew University of Jerusalem, 2002.

[22] P. Nayak and B. Williams. Fast context switching in real time propositional reasoning. In *Proceedings of 14th National Conference on Artificial Intelligence*, 1997.

[23] A. Otero. The SPHERES satellite formation flight testbed: design and initial control. Master's Thesis, MIT, 2000.

[24] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *The International Journal on Software Tools for Technology Transfer*, 7: 156-173, 2005

[25] P. Stephan, R. Brayton, and A. Sangiovanni-Vencentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 15: 1167-1176, 1996.

[26] B. Williams and R. Ragno. Conflict-directed A* and its role in model-based embedded systems. *To appear in the Journal of Discrete Applied Math (accepted 2001)*, 2006.

[27] G. Wu and G. Coghill. A propositional root antecedent ITMS. In *Proceedings of the 15th International Workshop on Principles of Diagnosis*, 2004.

[28] H. Zhang and M. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the 4th International Symposium on Artificial Intelligence and Mathematics*, 1996.

[29] H. Zhang and M. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24: 277-296, 2000.

[30] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, 2001.

[31] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Proceedings of the 8th International Conference on Computer Aided Deduction*, 2002.

4230-67