SEARCHING FOR TRUTH: TECHNIQUES FOR SATISFIABILITY OF BOOLEAN FORMULAS

Lintao Zhang

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

June 2003

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Sharad Malik
(Principal Adviser)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Margaret Martonosi

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Aarti Gupta

Approved for the Princeton University Graduate School:

Dean of the Graduate School

Abstract

The problem of determining whether a propositional Boolean formula can be *true* is called the Boolean Satisfiability Problem or SAT. SAT is an important and widely studied problem in computer science. In practice, SAT is a core problem in many applications such as Electronic Design Automation (EDA) and Artificial Intelligence (AI). This thesis investigates various problems in the design and implementation of SAT solvers.

Even though researchers have been studying SAT solving algorithms for a long time, efforts were mainly concentrated on improving the algorithms to prune search spaces. In this thesis, the implementation issues of SAT solvers are studied and some of the most widely used techniques are quantitatively evaluated. SAT solvers developed by utilizing the results of the analysis can achieve 10-100x speedup over existing SAT solvers.

For some applications, certain capabilities are required for the SAT solvers. For mission critical applications, the SAT solvers are often required to provide some means for a third party to verify the results. In the thesis a method to provide such a certification is discussed. The second capability discussed in the thesis is to find a small unsatisfiable sub-formula from an unsatisfiable SAT instance. This capability is useful for debugging purposes for some applications. The thesis discusses how these two capabilities are related and how to implement them in existing SAT solvers.

For some reasoning tasks, propositional formulas are insufficient. This thesis discusses the problem of deciding the satisfiability of Quantified Boolean Formulas

(QBF). A quantified Boolean formula contains universal and existential quantifiers and is more expressive than a propositional formula. A technique called learning and non-chronological backtracking, which has been shown to work very well on SAT solvers, is applied in a QBF solver. Experiments show that the technique, when properly adapted, can greatly improve the efficiency of the QBF solver.

This thesis is mainly about properly engineering satisfiability algorithms to solve practical problems. By carefully examining algorithms and implementation details, improvements have been made to enable SAT and QBF solvers to become more useful as feasible reasoning engines for real applications.

Acknowledgements

First of all, I would like to thank my advisor Professor Sharad Malik for his guidance and support throughout my graduate studies. His dedication to research and patience for students and colleagues will always be an example for me to follow. Without the numerous discussions and brainstorms with him, the results presented in this thesis would never have existed. I am grateful to Dr. Aarti Gupta, for her valuable comments and suggestions on my research. I would also like to thank Professor Margaret Martonosi for taking the time to read my thesis.

I am in debt to Yinlei Yu, Daijue Tang, Ying Zhao, Matthew W. Moskewicz and Conor F. Madigan for their fruitful discussions. I thank Dr. Pranav Ashar, Dr. Malay Ganai and Zijian James Yang, for their help during my collaboration with NEC. I would also like to thank all my friends in Princeton University, including Zhijie Shi, Xiao Yang, Zhen Luo, Xinying Zhang, Patrick McGregor, Yuan Xie, Hua Cao, Robert Dick, Zhining Huang, Rajagopalan Subramanian, Keith Vallerio, Vassos Soteriou, Han Chen, Limin Wang, and many others. They make the life at Princeton such a wonderful experience.

Last but not least, I would like to thank my parents for their constant love and support. It would be impossible for me to express my gratitude towards them in mere words. I dedicate this thesis to them.

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

1.1 The Satisfiability Problem 1

1.1.1 Automatic Reasoning and Its Applications 1

1.1.2 The Boolean Satisfiability (SAT) Problem 3

1.1.3 Beyond Propositional SAT: QBF Problem 5

1.2 Thesis Contributions 6

1.3 Thesis Organization 8

2 SAT Algorithms **10**

2.1 Basic Concepts for SAT Solving 10

2.1.1 Boolean Formula 10

2.1.2 Resolution and Consensus 14

2.2 A Brief Overview of SAT Solving 15

2.2.1 Resolution Based SAT algorithm 16

2.2.2 DLL Search Algorithm 18

2.2.3 Binary Decision Diagrams 22

2.2.4 Stålmarck's Algorithm 24

2.2.5 Stochastic Algorithms 27
2.3 Davis Logemann Loveland (DLL) Algorithm 30
2.3.1 The Basic Flow 31
2.3.2 Branching Heuristics 33
2.3.3 Deduction Engine 37
2.3.4 Conflict Analysis 39
2.3.5 Data Structure for Storing the Clause Database 46
2.3.6 Preprocessing, Restart and other techniques 48
2.4 Chapter Summary 49

3 Efficient Implementation of the DLL Algorithm 51
3.1 Benchmark Description 53
3.2 Boolean Constraint Propagation 56
3.2.1 Counter Based BCP 59
3.2.2 Pointer Based BCP 63
3.2.3 Experimental Results for Different BCP Schemes 67
3.3 Learning Schemes 70
3.3.1 Different Learning Schemes 72
3.3.2 Experimental Results 75
3.4 Cache Performances 78
3.4.1 Cache Performances of Different BCP mechanisms 80
3.4.2 Cache Performances of Different SAT Solvers 84
3.5 Chapter Summary 87

4 The Correctness of SAT Solvers and Related Issues 88
4.1 The Algorithm of a DLL SAT Solver 89
4.2 The Correctness of the DLL Algorithm 94
4.3 Resolution Graph 97

4.4 Checking the Unsatisfiable Proof 99
4.4.1 Producing the Trace for the Checker 101
4.4.2 The Depth-First Approach 103
4.4.3 The Breadth-First Approach 106
4.4.4 Experiment Results 107
4.5 Extracting Unsatisfiable Core 110
4.6 Related Work 122
4.7 Chapter Summary 123

5 Learning and Non-Chronological Backtracking in a QBF Solver 124
5.1 The QBF Problem 125
5.2 Solving QBF using DLL search: Previous Works 130
5.2.1 The Semantic Tree of QBF 131
5.2.2 DLL Search for QBF 133
5.2.3 Deduction Rules 136
5.3 Conflict-Driven Learning 141
5.3.1 The Top-Level Algorithm 141
5.3.2 Conflict Analysis in a QBF Solver 142
5.3.3 Long Distance Resolution 146
5.3.4 Stopping Criteria for Conflict Analysis 152
5.4 Satisfaction-Driven Learning 153
5.4.1 Augmented Conjunctive Normal Form (ACNF) 154
5.4.2 Implication Rules for the Cubes 158
5.4.3 Generating Satisfaction-Induced Cubes 159
5.4.4 Satisfaction Analysis in a QBF Solver 162
5.5 Experimental Results 165
5.6 Related Developments 172

5.7 Chapter Summary 172

6 Conclusions and Future Work **174**

6.1 Efficient SAT Solvers 174

6.2 Other SAT Related Issues 176

6.3 Beyond SAT: QBF and Other Research Directions 177

Bibliography **197**

List of Tables

3.1 SAT instances used in this section 58
3.2 BCP time as a percentage of total runtime for zchaff 58
3.3 Instances for evaluating different BCP schemes 69
3.4 Runtime for solver with different BCP mechanisms 70
3.5 Run time for different learning schemes (seconds) 77
3.6 Detailed statistics for three learning schemes 79
3.7 Instances used to evaluate the BCP mechanisms 81
3.8 Memory behavior of different BCP mechanisms: counter based 82
3.9 Memory behavior of different BCP mechanisms: pointer based 83
3.10 Instances for cache miss experiments for SAT solvers 85
3.11 Data cache miss rates for different SAT solvers 86
4.1 The unsatisfiable instances used for experiments 108
4.2 Trace generation and statistics of the resolution graph 109
4.3 Number of variables and clauses involved in the proofs 110
4.4 Extracting unsatisfiable cores 115
4.5 The quality of extracted unsatisfiable cores 118
4.6 Comparison of two core extraction methods: aim 120
4.7 Comparison of two core extraction methods: jnh 121
5.1 Runtime comparison of different QBF solvers Pg. 1 (unit: seconds) . 166
5.2 Runtime comparison of different QBF solvers Pg. 2 (unit: seconds) . 167
5.3 Runtime comparison of different QBF solvers Pg. 3 (unit: seconds) . 168

5.4 Detail Statistics of CDL and Full version of Quaffle 170

List of Figures

2.1 Search space of a SAT problem 19
2.2 The recursive description of DLL algorithm 21
2.3 An example of BDD 23
2.4 Local search algorithm of GSAT 28
2.5 The iterative description of DLL algorithm 32
2.6 An example to illustrate non-chronological backtracking 40
2.7 Implication graph 43
2.8 Cuts in an implication graph correspond to conflict clauses 44
2.9 Trie data structure for clause database 47
3.1 Four cases of Head/Tail BCP scheme 65
3.2 Four cases of 2-Literal Watching BCP scheme 66
3.3 Comparison of 2-literal watching scheme and Head/Tail List scheme . 68
3.4 Different learning schemes 71
4.1 The top-level algorithm of DLL with learning and non-chronological backtracking 90
4.2 Conflict analysis by iterative resolution 91
4.3 Learning as a resolution process 93
4.4 The resolution graph 98
4.5 The depth-first approach of the checker 104
4.6 Extract unsatisfiable core from the resolution graph 113
4.7 The Core extraction statistics as a ratio of the original 117

5.1 Some examples of QBF Instances 129
5.2 Semantic tree for QBF problem 132
5.3 The recursive description of the DLL algorithm for QBF solving . . . 135
5.4 The functions for deduction in QBF on a CNF database 140
5.5 The iterative description of the DLL algorithm for QBF solving . . . 143
5.6 Conflict analysis in a QBF solver 144
5.7 The function for deduction on an ACNF database 160
5.8 Satisfaction analysis in a QBF solver 163

Chapter 1

Introduction

In the process of pursuing higher productivity by utilizing the power of computers, several important computational tasks have been identified by researchers as the core problems in computer science. Among these tasks, a difficult and important one is the satisfiability problem. This thesis address some of the issues in solving this important problem for real world applications.

This chapter provides an overview of the problems that will be addressed in the rest of the thesis and gives a brief summary of the thesis contributions.

1.1 The Satisfiability Problem

1.1.1 Automatic Reasoning and Its Applications

The interest in using machines for logic deduction and reasoning can be traced back to the development of logic machines in the eighteenth and nineteenth centuries. Charles Stanhope's Stanhope Demonstrator [96], which was invented around 1777, is often regarded as the first real machinery to solve logic problems. William Stanley Jevons' famous Logic Machine, developed in 1869 [96], was the first machine that could solve a logic problem faster than a human. Since the invention of digital computers, much

effort has been dedicated to develop algorithms to solve logic problems efficiently utilizing the power of modern computers. This effort was first initiated by the Artificial Intelligence (AI) community to develop programs that have the ability to "think", and more recently has also been pushed by the problems resulting from the Electronic Design Automation (EDA) of digital systems.

Two important areas in electronic design automation are optimization and validation. On the optimization front, efforts are directed towards building circuits that are fast, small and energy efficient. On the validation front, the circuits need to be functionally correct. Automated logic reasoning is one of the core algorithms employed by both these tasks. Logic reasoning methods have been widely used in various parts of the design automation process such as logic optimization, test pattern generation, formal verification and functional simulation. Due to their practical importance, many automatic reasoning techniques have been studied in the literature. Numerous papers have been published on subjects such as Binary Decision Diagram (BDD) [15] and recursive learning [69]. Among various reasoning methods, one of the most important is the solving satisfiability.

Given a Boolean formula, the problem of determining whether there exists a variable assignment that makes the formula evaluate to *true* is called the satisfiability problem. If the formula is limited to only contain logic operations **and**, **or** and **not**, then the formula is said to be a *propositional Boolean formula*. Determining the satisfiability of a propositional Boolean formula is called the *Boolean Satisfiability Problem (SAT)*. If besides the three logic operations the formula also contains universal and existential *quantification*[1] over the variables, then the formula is said to be a *quantified Boolean formula*. The problem of determining the satisfiability of a quantified Boolean formula is called the *QBF problem*.

By solving the satisfiability problems, automatic tools can perform reasoning on

[1]The concept of quantification will be formally defined in Chapter 5.

Boolean formulas and digital circuits. The propositional Boolean Satisfiability (SAT) solvers have already been widely used in the EDA community and are gaining more ground due to recent advancements. Recently there has also been a lot of interest in the EDA community to solve the QBF problem efficiently. These two kinds of satisfiability solvers are the main topic of this thesis. They will be discussed briefly next, and in more detail in the following chapters.

1.1.2 The Boolean Satisfiability (SAT) Problem

The Boolean Satisfiability Problem (SAT) is one of the most important and extensively studied problems in computer science. Given a propositional Boolean formula, the SAT problem asks for an assignment of variables such that the formula evaluates to true, or a proof that no such assignment exists. SAT was the first problem shown to be NP-Complete [25]. In Electronic Design Automation, instances of problems reducible to SAT are ubiquitous, appearing in various applications such as Automatic Test Patten Generation (ATPG) [71, 125], combinational circuit equivalence checking [45, 86], sequential property checking [118], microprocessor verification [132], model checking [11, 91], reachability analysis [51, 2] redundancy removal [67], timing analysis [87] and routing [97], to name just a few.

The first SAT solving algorithm is often attributed to Davis and Putnam, who proposed an algorithm that can solve general SAT problems in 1960 [32]. Since then, numerous algorithms and techniques have appeared in the literature to improve the efficiency of SAT solving. Because of its NP-Complete nature, it is unlikely that there exist algorithms that can solve SAT in polynomial time in the size of the instance description (unless P=NP)[2]. However, unlike some "hard" NP-Complete problems (e.g. the quadratic assignment problem [36]), it is well known that many SAT instances can

[2]Many of the above mentioned problems in EDA have the same complexity as SAT (i.e. NP-Complete, e.g. see [38]).

be solved quite efficiently in practice. SAT instances with hundreds or even thousands of variables can often be solved by current state-of-the-art SAT solvers in seconds or minutes. Luckily, it seems that most of the SAT instances generated from practical EDA problems belong to this "easy" class of SAT instances [104]. In contrast, randomly generated SAT instances can be very hard [94]. In fact, some researchers have conjectured that no SAT solver can solve hard random 3-SAT instances with more than 700 variables within a reasonable amount of time [114, 76] (though recent work [35] seems to cast doubt on this conjecture).

However, SAT instances generated from real world applications can be huge. Current EDA tools regularly deal with designs with hundreds of thousands or even millions of gates. SAT instances generated from such circuits often contain hundreds of thousands or even more variables. Significant effort is needed to make SAT solvers efficient enough to attack these huge SAT instances. In fact, current SAT-based verification tools often spend a dominant part of their run time on SAT solving, which can easily be several hours for each instance. Making SAT solvers as efficient as possible is extremely important for practical purposes, as verification is now becoming the most time consuming task in digital integrated circuit (IC) design.

Among the numerous algorithms to solve SAT, the algorithms based on the search procedure proposed by Davis, Logemann and Loveland (DLL) [31] are gaining momentum and becoming the *de facto* algorithms to tackle SAT. During the last ten years or so, improvements proposed by various researchers have revolutionized the DLL algorithm by introducing some very effective techniques such as learning, non-chronological backtracking [88, 64] and restarting [46]. Advances in implementation and fine tuning [95, 135] also dramatically improved the efficiency of the SAT solvers. Current state-of-the-art SAT solvers, such as *zchaff* [138] developed at Princeton University, are capable of solving real world SAT instances with thousands, or sometimes even millions of variables in a reasonable amount of time. State-of-the-art SAT solvers

often outperform other automatic reasoning techniques in many applications [132]. Because of this, many industrial applications have started utilizing SAT solvers as the core deduction and reasoning engine and have seen much success [12, 27].

Even though people have been studying the DLL algorithm for many years, there is still a lot of insight that can be gained from careful examination of the algorithm. The speed of SAT solvers has been improving steadily over the last ten years and there are no signs of this stopping. New features such as incremental solving [126] and parallelization (e.g. [134]) have been successfully integrated into SAT solvers. Techniques have been proposed to utilize the basic algorithm of DLL to solve problems other than SAT (e.g. the pseudo-Boolean constraint problem [4], the min-cover problem [81]). In fact, the work described in this thesis is the consequence of a careful study of the DLL algorithm from various perspectives.

1.1.3 Beyond Propositional SAT: QBF Problem

Some problems generated from real world applications, such as model checking for Linear Temporal Logic (LTL), are shown to be PSPACE-Complete [121]. Therefore, they fall into a harder complexity category than NP-Complete problems such as SAT. It is unlikely that such problems can be formulated as SAT instances and solved by SAT solvers efficiently. Model checking has many real world applications in sequential circuit verification and protocol validation. Traditionally, Binary Decision Diagrams (BDDs) [15] are used as the deduction engine for such tasks. However, BDDs are well known to suffer from a memory explosion problem in many practical situations. Therefore, for a long time, model checking has seen limited use in industrial settings. Recently, the invention of search-based bounded model checking that relies on a SAT solver for deduction has led to much larger circuits that can be formally checked [11]. Unfortunately, SAT-based model checking is "bounded" in the sense that it can only verify the system's behavior for a "bounded" number of steps. Therefore, it can only

detect bugs in a design but cannot easily verify the design completely unless the necessary bound is known *a priori*. Due to the PSPACE-Complete nature of model checking, it is unlikely that a SAT solver is sufficient for the reasoning needs in model checking tasks. Therefore, it is necessary to investigate more powerful reasoning techniques than SAT.

A *Quantified Boolean Formula (QBF)* is a propositional formula with quantifiers preceding it. The task of determining whether a QBF is satisfiable is known to be a PSPACE-Complete problem [39]. It is well known that many model checking problems can be formulated as QBF problems efficiently. Therefore, if an efficient QBF solver exists, it is possible to use it to solve some model checking problems. Moreover, some applications in the Artificial Intelligence such as conditional planning [107] can also benefit from progress in QBF solving.

Unlike SAT, which has been under intensive investigation for the last forty years, QBF solvers have not seen much research effort until the last several years. As a result, current QBF solvers are far less sophisticated compared to SAT solvers. QBF is quite similar in nature to SAT, both in problem representation as well as in the search procedures. Not surprisingly, the DLL style search is applicable to QBF solvers as well. In this thesis, some of the techniques originally developed for SAT solvers are applied for QBF solving and performance improvements have been observed in many cases.

1.2 Thesis Contributions

A major part of this thesis tries to quantitatively analyze some of the most widely used techniques and heuristics in current state-of-the-art SAT solvers. The purpose is to understand the performance implications of employing these techniques in order to optimize the solver's performance in solving real world instances. The results of

the analysis are incorporated into the SAT solver *zchaff*, a highly optimized, state-of-the-art SAT solver that incorporates features such as learning and non-chronological backtracking, restarting and efficient decision heuristics.

As SAT solvers see increased industrial use, many mission critical applications (such as verification of the safety of train stations [47]) are beginning to utilize SAT solvers for the deduction and reasoning tasks. Because of this, it is very important to guarantee the correctness of the SAT solvers themselves. Modern SAT solvers are quite intricate and often employ many heuristics and algorithms. Therefore, they are not trivial to implement and more often than not contain various bugs. Some widely used SAT solvers have been found to be buggy after wide deployment. Therefore, for mission critical applications, a method is needed to independently check the validity of a SAT solver. In the thesis, a procedure is presented to accomplish this certifying process. Experiments show that the proposed procedure incurs very small performance penalty in practice.

For most of the SAT problems generated from real world applications, certain satisfiability results are often expected. For example, for instances generated from the equivalence checking of combinational circuits, the expected result is that the Boolean formula is unsatisfiable, which implies that the two circuits are equivalent. On the other hand, Boolean formulas generated from FPGA routing are expected to be satisfiable, which imply feasible routes. If a certain instance is expected to be unsatisfiable but the formula is actually satisfiable, then the satisfiable solution can act as a "stimulus" or "input vector" or "counter example" for debugging purposes. However, if the problem is expected to be satisfiable but found to be unsatisfiable, no useful debugging information can be gained from the fact that no "solution" exists. Traditionally, designers have to examine the entire Boolean formula in order to determine the reasons for unsatisfiability. In the thesis an automatic procedure that can efficiently extract a small unsatisfiable core from an unsatisfiable Boolean formula is

described. The unsatisfiable core extracted by the procedure enables the designer to concentrate on a smaller formula to debug the system. Unlike previous approaches to the problem, the procedure described is fully automatic. Moreover, it is much more efficient than previous approaches and can be applied on instances generated from real world applications.

Another contribution of this thesis is in improving the algorithms used for QBF. QBFs can be solved using a procedure similar to the DLL search used in SAT solving. In SAT solving, a technique called *non-chronological backtracking and learning* has been found to be very effective in improving the efficiency of the solvers. In the thesis, a similar technique is shown to be applicable to QBF solvers as well. It is shown that Conjunctive Normal Form (CNF) representation, the most widely used data structure for SAT solvers, is not sufficient to capture the information for QBF solving. A new data structure called the Augmented CNF is proposed. By using this data structure to represent the formula, a near symmetric view on both satisfying and conflicting leaves encountered during search can be achieved. Empirical results show that the algorithm proposed greatly improves the efficiency of the QBF solver on certain classes of benchmarks.

1.3 Thesis Organization

The organization of this thesis is as follows. Chapter 2 provides an introduction to the concepts and algorithms used in SAT solving. A brief introduction to various techniques used in SAT solvers is provided. In particular, the basic Davis Logemann Loveland (DLL) search algorithm is described in detail. The DLL procedure forms the foundation of most of today's SAT solvers, and is the main algorithm examined in the remainder of this thesis.

In Chapter 3, a quantitative evaluation of algorithms used for Boolean Constraint

Propagation (BCP) and learning is provided. These are two of the most important components of modern SAT solvers. The performance implications of different implementations on BCP and learning are analyzed. This chapter also investigates the cache behaviors of different BCP algorithms and different SAT solvers in order to gain some insight into cache friendly SAT solver design. The results obtained from the analysis are incorporated in a state-of-the-art SAT solver that has been widely used by many research groups.

The first half of Chapter 4 presents an independent checker to check the correctness of the algorithm used in the SAT solver described in Chapter 3. First the algorithm used in the SAT solver is proven to be sound and complete. This forms the basis for the checking procedure discussed. A procedure to verify the output of the SAT solver is provided and two different implementations of the algorithm are given. Experimental results are used to show the feasibility of the checker in practice. The second half of Chapter 4 deals with unsatisfiable core extraction. The concepts and results obtained in the first half of the chapter are used to derive an automatic unsatisfiable core extraction procedure. Experimental results are provided to show the efficiency and scalability of the procedure.

Chapter 5 presents the work on QBF solving. The notion of quantification is introduced and the similarities and differences between QBF solving and SAT solving are analyzed. In the rest of the chapter, the algorithms to incorporate learning and non-chronological backtracking are discussed step by step. Some empirical data are provided to validate the effectiveness of the proposed algorithm.

Finally, Chapter 6 contains a summary of the thesis and a discussion of some future research directions.

Chapter 2

SAT Algorithms

This chapter provides an overview of SAT solving algorithms. Section 2.1 introduces the basic concepts used in SAT solvers. Section 2.2 briefly reviews some of the algorithms widely used for SAT solving. These algorithms include Binary Decision Diagrams (BDDs), resolution based Davis Putnam (DP) algorithm, search-based Davis Logemann Loveland (DLL) algorithm, Stålmarck's algorithm and stochastic algorithms. Among them, the Davis Logemann Loveland (DLL) algorithm is the most widely used SAT algorithm and is the base for all the discussions found in this thesis. In Section 2.3 the details of solving SAT using DLL search are outlined. The topics discussed include the main DLL procedure as well as ideas such as non-chronological backtracking, learning and restarting.

2.1 Basic Concepts for SAT Solving

2.1.1 Boolean Formula

A *Boolean formula* is a string that represents a *Boolean function*. A Boolean function involve some Boolean *variables*. It maps the Boolean n-space B^n to Boolean space B, where n is the number of variables for the function and $B = \{True, False\}$ is the Boolean space. A Boolean variable takes only the two values in B. A Boolean

variable can be temporarily bound with a Boolean value (the variable is said to be *assigned* a value) or be *free*, which means that it is not assigned a value.

A *propositional* Boolean formula is a Boolean formula that contains only logic operations **and**, **or** and **not** (sometimes called *negation* or *complement*). The rest of this thesis will use F' to represent **not**(F); F_1F_2 or $F_1 \cdot F_2$ to represent F_1 **and** F_2, and $F_1 + F_2$ to represent F_1 **or** F_2. In the rest of this thesis, logic values *True* and *False* will be represented by 1 and 0 respectively. Using this notation, some examples of propositional Boolean formulas are listed below:

$$ab + (abc + a'c'd)(a + b')d$$

$$(x + y')(x + y + z')(x' + y)$$

$$abc + xyz + ac'x + d$$

Since only Boolean functions will be discussed in this thesis, the adjective "Boolean" may be omitted where no confusion may occur. In this chapter as well as the next two chapters (i.e., Chapters 3 and 4), if not indicated otherwise, all formulas discussed are propositional. Therefore, the adjective "propositional" may also be dropped in these three chapters.

A propositional Boolean formula can be expressed in *Conjunctive Normal Form (CNF)*, also known as *Product of Sum (POS)* form.

- A formula in CNF consists of a conjunction (logic **and**) of one or more *clauses*.
- Each clause is a disjunction (logic **or**) of one or more *literals*.
- A literal is an occurrence of a Boolean variable or its negation. If the literal is the occurrence of a variable, it is said to have the *positive polarity* (or is in *positive phase*). If it is the occurrence of the negation of a variable, it is said to have the *negative polarity* (or is in *negative phase*).

An example of a propositional formula in CNF is:

$$\varphi = (x + y + z')(x + x' + z')(x' + y + z')(x + y)(z)(z')$$

This formula contains six clauses: $(x + y + z')$, $(x + x' + z')$, $(x' + y + z')$, $(x + y)$, (z) and (z'). The first clause $(x + y + z')$ contains three literals, x, y and z'. For this particular example, the function that the formula φ represents is actually the constant 0, because of the last two clauses.

A clause is *redundant* in a CNF formula if deleting the clause from the CNF will not change the Boolean function the formula represents. Otherwise, the clause is said to be *irredundant*. For example, in the above example, the first four clauses are redundant because deleting any of them will not change the function φ represents. On the other hand, clause (z) is not redundant in φ. A clause C_1 is said to be *subsumed* by clause C_2 if all literals appearing in C_2 also appear in C_1. If a clause C in a CNF formula is subsumed by another clause in the formula, then clause C is redundant. In the above example, clause $(x' + y + z')$ is subsumed by clause (z'). A clause C is said to be a *tautology clause* if both a literal and its complement appear in C. In the above example, clause $(x + x' + z)$ is a tautology clause. A clause is said to be an *empty clause* if it does not have any literals in it.

Alternately, it is also possible to represent a Boolean formula in *Disjunctive Normal Form (DNF)*, also known as *Sum of Product (SOP) form*.

- A DNF formula consists of a disjunction (logic **or**) of one or more *cubes*;
- Each cube is a conjunction (logic **and**) of one or more literals.

An example of a propositional formula in DNF is:

$$\psi = xyz + xx'z + x'yz' + xy'z' + z + z'$$

This formula contains six cubes, xyz, $xx'z$, $x'yz'$, $xy'z'$, z and z'. The function represented by this formula is the constant 1 because of the last two cubes.

A cube is *empty* if the cube contains both a literal and its complement (e.g. $xx'z$ in ψ). A cube is *redundant* in a DNF formula if deleting the cube from the formula will not change the Boolean function the formula represents (e.g. the first four cubes in ψ). Otherwise, the cube is said to be *irredundant* (e.g. the last two cubes in ψ). A cube S_1 is said to be *contained* by another cube S_2 if any literal appearing in S_2 also appears in S_1 (e.g. in ψ, cube $xy'z'$ is contained by cube z'). Similar to a subsumed clause, a contained cube in a DNF is also redundant. A cube is said to be an *empty cube* if both a literal and its complement appear in the cube (e.g. $xx'z$ in ψ). If a cube contains no literal, then the cube is said to be a *tautology*.

An empty cube is redundant in DNF formulas, as are tautology clauses for CNF formulas. Therefore, if not pointed out otherwise, it is always assumed that the CNF formulas being discussed do not contain tautology clauses, and the DNF formulas do not contain empty cubes.

Given a propositional Boolean formula, the *Boolean Satisfiability Problem (SAT)* asks whether there exists a variable assignment that can make the formula evaluate to *true*. If such an assignment exists, the formula is *satisfiable*. Otherwise, the formula is *unsatisfiable*. A problem related to SAT is the *tautology problem*. For a given propositional formula, if *all* variable assignments make it evaluate to *true*, then the formula is a *tautology*. Given a formula, the tautology problem asks whether it is a tautology. Obviously, if a formula is a tautology, then its negation is unsatisfiable, and vice versa.

It is easy to determine whether a formula in CNF is a tautology: a CNF formula is a tautology if and only if all of its clauses are tautology clauses. On the other hand, determining if a CNF formula is satisfiable is an NP-Complete problem [25]. In contrast, determining whether a DNF formula is satisfiable is trivial, while determining if it is tautology is Co-NP-Complete. There is no polynomial algorithm that can convert a CNF formula into an equivalent DNF formula and vice versa unless

$P = NP$. However, given a propositional formula of any form, it is easy to construct a CNF formula that has the same satisfiability by introducing new variables (a structural preserving method is given by Plaisted and Greenbaum [103]). The construction takes linear time and the resulting formula is linear in the size of the original formula. The constructed CNF is satisfiable if and only if the original formula is satisfiable. Therefore, it is sufficient for a SAT solver to be able to evaluate CNF formulas only. Moreover, some nice properties of a CNF formula make it especially suitable for SAT solving. These properties will be discussed later in this chapter. Currently almost all modern SAT solvers limit the input to be in CNF. In future discussion, if not explicitly pointed out otherwise, the propositional formula is always assumed to be presented in Conjunctive Normal Form.

2.1.2 Resolution and Consensus

If S is a set of literals, $\sum S$ will be used to represent the disjunction of all the literals in S and $\prod S$ will be used to represent the conjunction of all the literals in S. The consensus law states that it is possible to generate a redundant clause (cube) from two clauses (cubes) if certain conditions are met (see, e.g. [53]).

Proposition 1: *Consensus Law:*

If S_1 and S_2 are two sets of literals, and x is a literal then:

$$(x + \sum S_1)(x' + \sum S_2) \leftrightarrow (x + \sum S_1)(x' + \sum S_2)(\sum S_1 + \sum S_2)$$

$$(x \prod S_1) + (x' \prod S_2) \leftrightarrow (x \prod S_1) + (x' \prod S_2) + (\prod S_1 \prod S_2)$$

For example, if a, b and c are literals, then:

$$(a + b)(a' + c) \leftrightarrow (a + b)(a' + c)(b + c)$$

and

$$ab + a'c \leftrightarrow ab + a'c + bc$$

The operation of generating clause $(\sum S_1 + \sum S_2)$ from clauses $(x + \sum S_1)$ and $(x' + \sum S_2)$ is called *resolution*. The resulting clause $(\sum S_1 + \sum S_2)$ is called the *resolvent* of clause $(x + \sum S_1)$ and clause $(x' + \sum S_2)$. In our example, clause $(b + c)$ is the resolvent of clauses $(a + b)$ and $(a' + c)$. The operation of generating cube $\prod S_1 \prod S_2$ from cubes $x \prod S_1$ and $x' \prod S_2$ is called *consensus*. The resulting cube is called the *consensus cube* of the two original cubes. The above theorem shows that the resolvent of two clauses is redundant with respect to the original clauses, as is the consensus of two cubes with respect to the original cubes.

The *distance* between two sets of literals S_1 and S_2 is the number of literals l such that $l \in S_1$, $l' \in S_2$. In the above theorem, if the distance between S_1 and S_2 is larger than 1, then the consensus of cubes $x \prod S_1$ and $x' \prod S_2$ is an empty cube, and the resolvent of clauses $(x + \sum S_1)$ and $(x' + \sum S_2)$ is a tautology clause. Because of the redundancy of tautology clauses and empty cubes, if not pointed out otherwise, resolution and consensus are usually limited to two clauses/cubes with distance 1 only. Here, the distance between two clauses C_1 and C_2 is defined as the distance between literal sets S_1 and S_2 where $S_1(S_2)$ is the set of literals appearing in clause C_1 (C_2). The distance between two cubes is similarly defined.

2.2 A Brief Overview of SAT Solving

Because of the practical importance of Boolean Satisfiability, significant research effort has been spent on developing efficient SAT solving procedures for practical purposes. The research has resulted in the development of many SAT algorithms that have seen practical success. Gu *et al.* [50] provide an excellent review of many of the algorithms developed thus far. In this section, a brief description is provided for some of the often-used algorithms for efficient SAT solving. Here only the more widely used techniques that have seen practical success are covered. Due to the extent of SAT

related material in the literature, it is almost impossible to have an extensive review of all the related work.

Using machines to solve logic problems can be dated back to William Stanley Jevon's Logic Machine in 1869. Quine's iterated consensus proposed in 1952 is a complete algorithm for tautology checking, which can be regarded as a general procedure for solving the dual of the SAT problem. Even though there were many developments pre-dating them, the original algorithm for solving SAT is often attributed to Davis and Putnam for proposing a resolution-based algorithm in 1960 [32]. The original algorithm proposed suffers from the problem of memory explosion. To cope with this problem, Davis, Logemann and Loveland [31] proposed a modified version in 1962 that used search instead of resolution to limit the memory footprint of the algorithm. This algorithm is often referred to as the DLL or DPLL algorithm while the original resolution-based algorithm is referred to as the DP algorithm.

Besides these two algorithms, other algorithms for SAT solving have also appeared in literature. These algorithms are based on various principles. Among them, the most successful ones are Binary Decision Diagrams [15], Stålmarck's algorithm [124, 119] and local search and random walk [49, 115]. These techniques have their respective strength and weaknesses, and all have been successfully applied in various application domains to attack the SAT problem. This section briefly discusses these methods.

2.2.1 Resolution Based SAT algorithm

The algorithm for SAT solving proposed by Davis and Putnam is a resolution-based algorithm [32]. The operation of resolution is defined in Section 2.1.2. Two clauses can be resolved to generate a new clause if their *distance* is 1. The resulting clause (called the *resolvent* clause) contains all the literals that appear in the two original clauses except the literal that appears in both with different phases.

Given a CNF formula and a variable x, the clauses in the formula can be divided into three sets:

1. clauses that do not contain x will be denoted as the clause set A;
2. clauses that contain x in positive phase will be denoted as the clause set B;
3. clauses that contain x in negative phase will be denoted as the clause set C;

Given two clauses - one from set B and one from set C, a resolvent clause can be generated that has no variable x in it. Davis and Putnam showed that if all such possible resolvent clauses between sets B and C are generated, the tautology clauses among them are deleted, and the remaining ones are conjuncted with the clauses in set A, then the resulting CNF formula will have the same satisfiability as the original formula with one fewer variable.

Besides resolution, the DP algorithm introduces a rule called *unit literal rule* for variable elimination. The rule states that if there exists a clause in the CNF formula that only contains one literal, then the formula resulting from replacing that literal with value 1 has the same satisfiability as the original formula.

Davis and Putnam also introduced another variable elimination rule called the *pure literal rule*: A variable (or literal) is said to be *pure* if it only occurs in a single phase in the CNF formula. The pure literal rule states that given a Boolean formula in CNF, the formula resulting from deleting all clauses that contain any pure variables (or literals) from it has the same satisfiability as the original formula.

The DP algorithm applies the unit literal rule and pure literal rule on the CNF formula until the formula cannot be simplified further. Then it applies the resolution process to eliminate one variable from the formula and again tries to apply the two rules on the resulting formula. This process is carried on iteratively on a CNF formula and eliminates variables one by one. At the end of the iteration process, the formula

will contain no variables. If the resulting formula contains an empty clause (clause that does not contain any literals), then the original formula is unsatisfiable. If the resulting formula is an empty formula (formula that contains no clause), then the original formula is satisfiable.

In the Davis Putnam algorithm, each time a variable is eliminated by resolution, in the worst case the number of clauses in the formula may grow quadratically. Therefore, the worst case memory requirement for Davis Putnam algorithm is exponential. In practice the DP algorithm can only handle SAT instances with tens of variables because of this memory explosion problem. Recently, Chatalic and Simon [23, 24] proposed the use of Zero suppressed Binary Decision Diagrams (ZBDDs) [93] to represent the clauses in the CNF formula. ZBDDs can often reduce the memory needed to represent the clauses by sharing common parts among clauses. Their experiments show that for certain classes of SAT instances, the DP algorithm using ZBDDs to represent clauses can perform very well compared with other SAT algorithms.

2.2.2 DLL Search Algorithm

The search algorithm proposed by Davis, Logemann and Loveland [31] is by far the most widely studied algorithm for SAT solving. Unlike the traditional DP algorithm and some other SAT solving algorithms discussed in the subsequent subsections (e.g. Binary Decision Diagram [15]), the memory requirement for the DLL algorithm is usually predictable. Therefore, SAT solvers based on DLL can handle very large formulas without memory overflow, and the solvers are only run-time limited. This property is usually desirable in EDA applications, since the majority of the SAT instances are large but relatively easy to solve.

The DLL algorithm is a branch and search algorithm. The search space is often presented as a binary tree shown in Figure 2.1. In the tree, each node represents the Boolean formula under a variable assignment. The leaf nodes represent complete

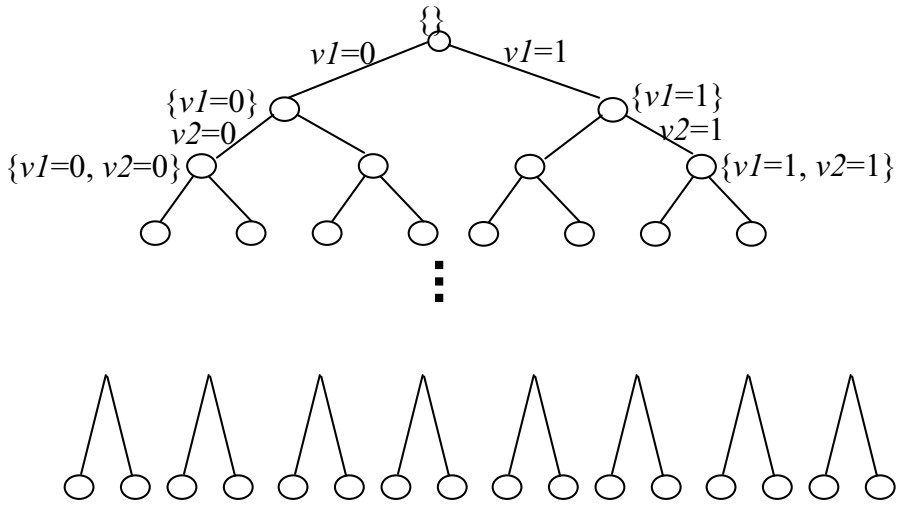


Figure 2.1: Search space of a SAT problem

assignments (i.e., all variables are assigned) while internal nodes represent partial assignments (i.e. some variables are assigned, the rest are free). If any of the leaf nodes have a valuation of true, then the formula is satisfiable. The DLL search procedure tries to search the entire tree and determine if such a true leaf node exists. If a true leaf node is found, the solver will produce the path from the root to the node, which is the variable assignment that satisfies the formula; otherwise, the solver will report that the formula is unsatisfiable.

Obviously, some pruning techniques are needed in order to avoid explicit exponential enumeration of all the leaves in DLL search. In almost all DLL-search-based algorithms, the formulas are required to be in CNF. A CNF formula is satisfied if and only if each of its clauses is satisfied individually. A clause is satisfied if and only if at least one of its literals evaluate to 1. If a clause is not satisfied, it is *unsatisfied*. From this, two important rules for DLL search are drawn [31]:

- *Unit Literal Rule*: If an unsatisfied clause has all but one of its literals evaluate to 0, then the remaining free literal must evaluate to 1 to make the clause (and

the entire CNF formula) satisfiable. Such clauses are called *unit clauses* and the free literal is called the *unit literal*.

- *Conflicting Rule*: If a partial assignment makes all literals of a clause evaluate to 0, then no satisfying leaf exists for the sub-tree under this partial assignment and the solver needs to backtrack and search other branches in order to find satisfying leaves. A clause that has all 0 literals is called a *conflicting clause*.

When a unit literal is forced to be assigned a value by a unit clause due to the unit literal rule, the literal is *implied*. The task of implying unit literals is called *unit implication*. The unit clause is called the *antecedent* of the variable corresponding to the implied literal. The process of iteratively assigning all unit literals to the value 1 till no unit clause left is called *Boolean Constraint Propagation (BCP)*(e.g. [88]).

BCP and backtracking constitute the core operations of a DLL solver, as shown in the next section. These two simple rules, together with some other techniques make DLL a very powerful procedure. Most (if not all) current state-of-the-art SAT solvers are based on the DLL procedure.

Traditionally the DLL algorithm is presented as a recursive procedure. The pseudo-code for this is shown in Figure 2.2 . Function `DLL()` is called with a CNF formula and a set of variable assignments. Function `deduction()` will return with a set of the necessary variable assignments that can be deduced from the existing variable assignment by the unit literal rule. These assignments are added to the original variable assignment as the new current set of assignments. The formula will be evaluated under the new current assignment.

If the formula is either *satisfied* (i.e. evaluates to 1 or *true*) or *conflicting* (i.e. evaluates to 0 or *false*) under the current variable assignment, the recursion will end and the result will be reported. Otherwise, the algorithm will make a branch. It chooses an unassigned variable called a *branching variable* from the formula, assigns it

```
DLL(formula, assignment)
{
    necessary = deduction(formula, assignment);
    new_asgnmnt = union(necessary, assignment);
    if (is_satisfied(formula, new_asgnmnt))
        return SATISFIABLE;
    else {
        if (is_conflicting(formula, new_asgnmnt))
            return CONFLICT;
    }
    branching_var = choose_free_variable(formula, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(branching_var, 1));
    result1 = DLL(formula, asgn1);
    if (result1 == SATISFIABLE)
        return SATISFIABLE;
    asgn2 = union (new_asgnmnt, assign(branching_var, 0));
    return DLL(formula, asgn2);
}
```

Figure 2.2: The recursive description of DLL algorithm

with a value of either true or false, appends this assignment to the current assignment and calls `DLL()` recursively. If this branch fails (i.e. it evaluates to *false*), then the solver will undo the branch and try the other value for the branching variable. If both branches fail, then the formula is unsatisfiable. The solution process begins by calling the function `DLL()` with an empty set of variable assignments.

The recursive procedure of DLL is often not efficient in practical implementations. In Section 2.3, an alternative formulation of the DLL search will be presented and the procedures involved in the solving process will be discussed in detail.

2.2.3 Binary Decision Diagrams

Bryant proposed the use of Reduced Ordered Binary Decision Diagrams (ROBDDs, often abbreviated as BDDs) as a canonical representation for logic function manipulation in 1986 [15]. Since then BDDs have been successfully used in many aspects of EDA and are regarded as one of the most efficient data structures for manipulating Boolean functions. There are numerous works aimed at improving the efficiency of BDDs (e.g. efficient implementations [13], dynamic variable re-ordering [111]) and expanding BDDs to other domains (e.g. ADD [106], BMD [17]). For a review of BDDs and other related decision diagrams, readers are referred to a good survey paper [16].

A BDD is a directed acyclic graph (DAG) that can be used to represent Boolean functions. In a BDD, nodes represent Boolean functions. Two kinds of nodes can appear in a BDD: *terminal* nodes and *regular* nodes. There are two terminal nodes, representing the logic functions *true* and *false* respectively. Terminal nodes have no outgoing edges. Each regular node in a BDD is associated with a variable and has two outgoing edges. If the internal node represents function f, and is associated with variable x, then the outgoing edges of this node point to nodes that represent functions $f(x = 1)$ and $f(x = 0)$ respectively. In a BDD, each variable has a distinct order number. A BDD is ordered, which means that all paths from a regular node

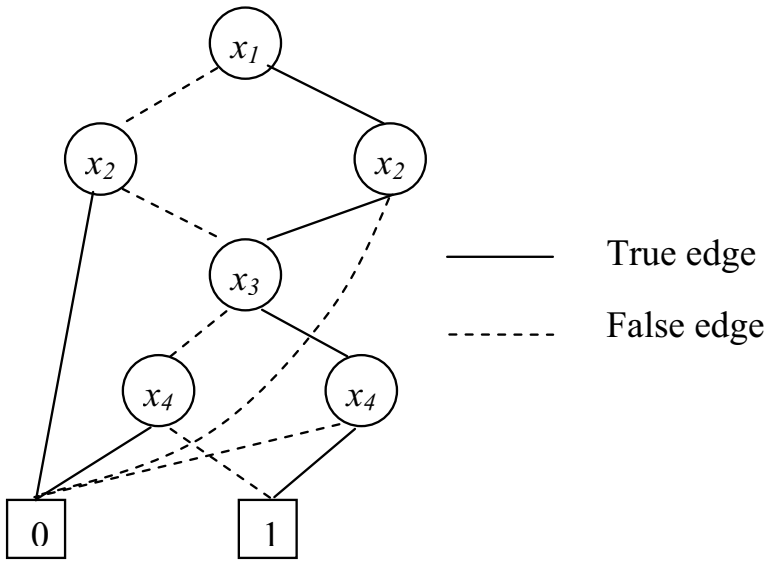


Figure 2.3: A simple BDD representing the function $(x_1 \Leftrightarrow x_2) \wedge (x_3 \Leftrightarrow x_4)$

to a terminal node must follow the variable ordering. A BDD is also reduced in the sense that no node has both of its outgoing edges pointing to the same node, and no two nodes have the same sub-graphs. An example of a BDD is shown in Figure 2.3.

A BDD is a *canonical* representation for Boolean functions. Two BDDs represent the same Boolean function if and only if the two graphs are isomorphic. To check whether a Boolean formula is satisfiable, one only needs to build the BDD representation of the formula and check whether it is isomorphic to the graph representing constant *false*. If not, then the formula is satisfiable, otherwise it is unsatisfiable. On modern BDD implementations [13], the isomorphic checking is a constant time operation.

Unfortunately, given a Boolean function, the task of building its BDD representation is often not trivial. The size of the BDD for a given function is often extremely sensitive to the variable ordering. For example, the BDD representation of an adder can be either linear or exponential in the size of the adder's bit-width, depending on

the variable ordering. Even though sometimes it is possible to find relatively good variable ordering using heuristics (e.g. [80]), the result is not guaranteed. Computing the optimal variable ordering for BDD is intractable (e.g. [33]). Moreover, it is known that the BDD representation for certain Boolean functions (e.g. the Boolean function that represents the most significant output bit of a multiplier as a function of the inputs) is exponential in size regardless of the variable order [15]. For these reasons, in practice the Boolean functions that BDDs can represent are often limited to less than several hundred variables in size. On the other hand, once the BDD is built, it can provide much more information about the Boolean function than its satisfiability; therefore, using a BDD for a SAT check is usually overkill.

2.2.4 Stålmarck's Algorithm

Stålmarck's algorithm [124] is a tautology checking algorithm proposed by Gunnar Stålmarck in the early 1980's. The algorithm is patented and a commercial solver based on this algorithm is available from Prover Technology [128]. To check if a Boolean formula is satisfiable, it is only necessary to check whether its complement is a tautology. Therefore, Stålmarck's algorithm can be used for SAT solving as well. There are several slightly different presentations of Stålmarck's algorithm in literature. The following presentation of the algorithm mainly follows a paper by Sheeran and Stålmarck [119].

Stålmarck's algorithm works on a data structure called a *triplet*. A triplet (p, q, r) represents the logic relationship[1]:

$$p \leftrightarrow (q \rightarrow r)$$

where p, q, and r are literals. Any logic function can be represented using only

[1] In this subsection, standard notation will be used to represent logic operations: $\wedge$(**and**), $\vee$(**or**), $\neg$(**not**), $\leftrightarrow$(**equivalent**) and $\rightarrow$(**imply**).

implication ("$\rightarrow$") and logic value *false*, for example:

$$A \wedge B \Leftrightarrow \neg(A \rightarrow \neg B)$$

$$A \vee B \Leftrightarrow \neg A \rightarrow B$$

$$\neg A \Leftrightarrow A \rightarrow false$$

Therefore, a logic function can be transformed into a set of triplets by introducing new variables to represent each logic connection. For example, the formula $p \rightarrow (q \rightarrow p)$ can be represented as two triplets $(b1, q, p)$, $(b2, p, b1)$ where b2 represents the formula itself.

Several simple rules can be used to simplify triplets and deduce new results. Some of the simple rules are listed here:

$$(0, y, z) : y = 1, z = 0;$$

$$(x, y, 1) : x = 1;$$

$$(x, 0, z) : x = 1;$$

$$(x, 1, z) : x = z;$$

$$(x, y, 0) : x = y';$$

$$(x, x, z) : x = 1, z = 1;$$

$$(x, y, y) : x = 1;$$

There are three triplets that represent contradiction in the logic; they are called terminal triplets. These triplets are $(1, 1, 0)$, $(0, x, 1)$ and $(0, 0, x)$. When one of these terminal triplets appears in the formula, the formula can be declared contradictory. If that happens, then the formula cannot be false, thus proving it to be a tautology. On the contrary, if the formula is assumed to be true and a terminal triplet is found,

then the formula is unsatisfiable. The algorithm starts by assuming the formula to be false and tries to derive a terminal triplet.

The simple rules are not complete. Given a formula, using the simple rules by themselves cannot guarantee to find out the satisfiability of the formula. Stålmarck's algorithm relies on another rule called the *dilemma rule* for branching and search. The first level of the dilemma rule chooses one variable from the formula, assumes it to be *true* and uses the simple rules to deduce a set of results. It then assumes the variable to be *false* to deduce another set of results. If one branch fails (i.e. it contains a contradiction), then the variable must take the other value. Otherwise, the intersection of these two sets must always be true for the formula, so it can be added back to the original formula. Doing this for all variables, the solver is said to achieve *1-saturation*. If 1-saturation cannot prove the tautology of the formula, the solver can continue with *2-saturation* by choosing two variables at a time, assigning four combinations of their valuation and extracting the common part. If that also fails, the solver can go on with 3-saturation and so on. Given a formula with n variables, Stålmarck's algorithm needs at most n-saturation to determine the satisfiability of it.

In practice, most formulas can be checked by 1-saturation or 2-saturation. Therefore, in most cases Stålmarck's algorithm can finish quickly without exponential search. Because of its proprietary nature, SAT solvers based on Stålmarck's algorithm are not widely available to the public and few evaluations have been carried out to compare its performance with SAT solvers based on DLL search. The only publicly available SAT solver that uses algorithms similar to the Stålmarck's algorithm is HeerHugo [48]. Some experimental results suggest that the solvers based on Stålmarck's algorithm compare favorably with SAT solvers based on DLL search [6].

2.2.5 Stochastic Algorithms

All of the algorithms and methods for solving SAT described in previous sections are *complete*. Given enough run time (and memory), a complete SAT algorithm can always find a solution for a SAT instance or prove that no solution exists. *Stochastic* SAT algorithms are a different class of SAT algorithms based on mathematical optimization techniques. Stochastic methods cannot prove a SAT instance to be unsatisfiable, but given a satisfiable instance, a SAT solver based on stochastic methods may find a solution with enough run time. Some applications such as AI planning [65, 66] and FPGA routing [97] can utilize stochastic SAT methods because in those applications instances are likely to be satisfiable and proving unsatisfiability is not required. For certain class of benchmarks such as hard random SAT, stochastic methods can significantly outperform existing complete algorithms that are based on systematic search (e.g. DLL algorithm). In this section, some stochastic methods for SAT solving are briefly discussed.

The SAT problem can be regarded as a special case of combinatorial optimization problems. The basic idea of optimization based stochastic methods for SAT solving is to regard the number of unsatisfied clauses as the objective function, and try to minimize this objective by assigning values to variables. Many heuristics and algorithms developed for general combinatorial optimization can also be applied to SAT solving. Over the years, researchers have experimented with various optimization techniques for SAT such as simulated annealing [122], genetic algorithms [83], neural networks [123], tabu search [89] and local search [49, 115]. Among these optimization methods, it seems that local search based algorithms have the best performance for SAT, and are most widely studied in literature.

According to Hoos [58], local search was first proposed independently by Selman, Levesque and Mitchell [115] and Gu [49]. Since then, numerous improvements and variations have been proposed (e.g. [59, 113]). Some theoretical results about local

```
GSAT_Solver() {
    for i = 1 to MAX_TRIES
    {
        T = a randomly generated assignment
        for j = 1 to MAX_FLIPS
        {
            if (no unsatisfied clauses exist)
                return T
            else
                x = find variable in T with max score
                if (score of x is smaller than 0)
                    break;
                else
                    flip the assignment of x
            }
        }
        return No Satisfying assignment found
    }
}
```

Figure 2.4: Local search algorithm of GSAT

search have also been obtained [100, 29]. Local search is based on the concept of hill climbing. The search space is the Boolean space of assignment combinations for all the variables. Given a variable assignment, the basic action of a local search algorithm is to try a move from the assignment in order to reduce the objective function or to escape from a local minimum. A move in most of the algorithms is just a flip of an assigned variable. Therefore, the Hamming distance of two assignments before and after a move is usually one. Various local search algorithms differ mainly on how to choose the variable to flip and how to escape from the local minimum. GSAT [115] and WalkSAT [90] are two representative local search algorithms.

- **GSAT**

  GSAT is the algorithm proposed by Selman, Levesque and Mitchell in 1992 [115].

It is basically a simple greedy algorithm to minimize the objective function. The algorithm of GSAT is described in Figure 2.4. It begins by choosing a random variable assignment. In each search step, if the instance is not satisfied under the current assignment, then the algorithm chooses a variable with the maximum score and flips it. The score of a variable x is the difference between current number of unsatisfied clauses and the number of unsatisfied clauses if x is flipped. If several variables have the same score, a random one among them will be chosen. If no variable has a positive score, or max flip steps in a round have been reached, then the algorithm will escape this local minimum with a new round by restarting the solving with a random variable assignment. If the maximum number of rounds has been reached without finding a satisfying solution, the algorithm will abort.

- **WalkSAT**

  WalkSAT was proposed by McAllester, Selman and Kautz in 1997 [90]. The main difference between WalkSAT and GSAT is in the selection of the variables to be flipped. In WalkSAT, a variable's score is the number of clauses that will change from satisfied to unsatisfied if that variable is flipped. Notice this is different from the score definition of GSAT. If the instance is not satisfied, the algorithm will first randomly choose a clause from the unsatisfied clauses. If there exists a variable in the selected clause with score 0 (i.e., flipping it will not make any satisfied clauses become unsatisfied), then the variable will be flipped. Otherwise, with a certain probability, one of the variables in the clause with the highest score will be flipped. This is called a random walk. In the remaining case, a random variable will be chosen from the clause and flipped.

  In practice, WalkSAT often outperforms GSAT because the noise introduced in random walk is often sufficient for the search to escape from local minima (in

contrast, GSAT has to restart). A state-of-the-art solver based on the WalkSAT principle is described by Hoos [59].

Besides random walk, other methods for escaping from local minima have also been proposed. For example, Shang and Wah [117] propose to weight clauses such that clauses that are often unsatisfied have higher weights. Such clauses are forced to be satisfied first. This method, sometimes called Dynamic Local Search, has also been shown to be successful in practice [113].

There exist some efforts to combine local search SAT algorithms with a complete DLL based algorithm. For example, Ginsberg and McAllester [40] have combined GSAT and a dynamic backtracking scheme. Hirsh and Kojevnikov [56] proposed a solver called UnitWalk that combines local search and unit clause elimination. Habet *et al.* proposed a hybrid approach [52] that combines WalkSAT and satz [75], a solver based on the DLL algorithm. Some researchers challenged the research community to develop a combined local search and systematic search algorithm that can outperform both in most SAT benchmarks [114]. It seems that this goal has not been achieved so far.

2.3 Davis Logemann Loveland (DLL) Algorithm

The Davis, Logemann, Loveland (DLL) [31] algorithm is the most widely used and studied algorithm for SAT. It also forms the foundation for most of the results in this thesis. In this section, the DLL algorithm is discussed in more detail. This section will first give an overview of the main flow, and then describe each of the functions in the flow.

2.3.1 The Basic Flow

As described in Section 2.2.2, traditionally the DLL algorithm is often presented in a recursive manner. However, this is usually not the way DLL SAT solvers are implemented. Marques-Silva *et al.* [88] generalized many of the actual implementations of various solvers based on DLL and rewrote it in an iterative manner as shown in Figure 2.5. The algorithm described in Figure 2.5 is an improvement of recursive algorithm in Figure 2.2 because it allows the solver to backtrack non-chronologically, as described in the rest of this section.

The algorithm described in Figure 2.5 is a branch and search algorithm. Different solvers based on DLL differ mainly in the detailed implementation of each of the functions shown in Figure 2.5. The framework of Figure 2.5 will be used as the foundation for the discussions on DLL that follow.

When the solving process begins, all variables are *free* (i.e. not assigned a value). In the beginning of the algorithm the solver will perform some preprocessing on the instance, performed by function `preprocess()`, to find out if the formula's satisfiability can be trivially determined or if some assignments may be made without any branch.

If preprocessing cannot determine the outcome of the formula's satisfiability, the main loop begins. A branch is made on a free variable by assigning it with a value. This operation is called a *decision* on a variable. The variable selected for branching is called the *decision variable*, and it will have a *decision level* associated with it. The decision level starts from 1 and is incremented with subsequent decisions. This is performed by function `decide_next_branch()`. After the branch, the problem is simplified as a result of this decision and its consequences. The function `deduce()` performs some reasoning to determine variable assignments that are needed for the problem to be satisfiable given the current set of variable assignments. Variables that are assigned as a consequence of the deduction after a certain branch will assume

```
DLL_iterative()
{
    status = preprocess();
    if (status!=UNKNOWN)
        return status;
    while(1) {
        decide_next_branch();
        while (true)
        {
            status = deduce();
            if (status == CONFLICT)
            {
                blevel = analyze_conflict();
                if (blevel < 0)
                    return UNSATISFIABLE;
                else
                    backtrack(blevel);
            }
            else if (status == SATISFIABLE)
                return SATISFIABLE;
            else break;
        }
    }
}
```

Figure 2.5: The iterative description of DLL algorithm

the same decision level as the decision variable. If a variable is assigned during the preprocessing, because no decision exists yet, the variable will assume a decision level of 0.

After the deduction, if all the clauses are satisfied, then the instance is satisfiable; if there exists a *conflicting clause* (i.e a clause with all its literals evaluating to 0), then the current branch cannot lead to a satisfying assignment, so the solver will backtrack. If the instance is neither satisfied nor conflicting under the current variable assignment, the solver will choose another variable to branch and repeat the process.

When a conflict occurs in the search process, the solver will *backtrack* and undo some of the branches. Which decision level to backtrack to is determined by the function `analyze_conflict()`. Backtracking to a decision level smaller than 0 indicates that even without any branching the instance is still unsatisfiable. In that case, the solver will declare the instance to be unsatisfiable. Within the function `analyze_conflict()`, the solver may perform some analysis and record some information from the current conflict in order to prune the search space for the future.

Different SAT solvers based on DLL differ mainly on the implementation of the various functions in the top level algorithm. In the next several subsections, some of the most successful implementations of these functions will be discussed.

2.3.2 Branching Heuristics

Branching occurs in the function `decide_next_branch()` in Figure 2.5. When no more deduction is possible, the function will choose one variable from all the free variables and assign it a value. After this assignment, the formula can be simplified. The importance of choosing good branching variables is well known. Different branching heuristics may produce drastically different sized search trees for the same basic algorithm, thus significantly affect the efficiency of the solver. Over the years

many different branching heuristics have been proposed. Not surprisingly, comparative experimental evaluations have also been carried out by various authors (e.g. [34, 84]).

Early branching heuristics such as Bohm's Heuristic (reported in [20]), Maximum Occurrences on Minimum sized clauses (MOM) (e.g. [37]), and Jeroslow-Wang [61] can be regarded as greedy algorithms that try to make the next branch generate the largest number of implications or satisfy most clauses. All these heuristics use some functions to estimate the effect of branching on each free variable, and choose the variable that has the maximum function value. One of the more successful branching heuristics based on statistics is introduced in SAT solver GRASP [84]. The scheme propose the use of the counts of literals appearing in unresolved (i.e. unsatisfied) clauses for branching. In particular, it was found that the heuristic that chooses the variable with the Dynamic Largest Combined Sum (DLIS) of literal counts in both phases gives quite good results for the benchmarks tested.

Notice that in the DLIS case, the counts are *state-dependent* in the sense that different variable assignments will give different counts for the same formula (represented by a *clause database* in the SAT solver). Whether a clause is unresolved (unsatisfied) or not depends on the variable assignment. Because the counts are state-dependent, each time the function `decide_next_branch()` is called, the counts for all the free variables need to be recalculated. This often introduces significant overhead.

Most of the heuristics mentioned above work well for certain classes of instances. However, all of these branching heuristics are based on the statistics of the formula being solved such as clause length, literal appearance frequency, and so on. Decision strategies based on these statistics, though often relatively effective for random SAT instances, usually do not capture enough information about SAT instances generated from real world applications[2]. Moreover, these statistics are *static* in the sense that

[2]Instances generated from real world applications are often called *structured* instances, in contrast

they are not aware of the search progress of the SAT solver. Because of the static nature, totally state-independent statistics are usually not good for branching purposes. Therefore, due to the state-dependent nature, the effective branching heuristics usually introduce a large overhead in the solver.

As deduction techniques become more and more efficient, the time spent on calculating statistics for branching begins to dominate the run time. Therefore, more efficient and effective branching heuristics are needed. The authors of the SAT solver chaff [95] proposed a branching heuristic called Variable State Independent Decaying Sum (VSIDS). VSIDS keeps a score for each phase of a variable. Initially, the scores are the number of occurrences of a literal in the initial formula. Because modern SAT solvers have a mechanism called *learning*[3], additional clauses are added to the clause database as the search progresses. VSIDS increases the score of a literal by a constant whenever an added clause contains the literal. Moreover, as the search progresses, periodically all the scores are divided by a constant number. In effect, the VSIDS score is a literal occurrence count with higher weight on the more recently added clauses. VSIDS will choose the free variable that corresponds to the literal with the highest score to branch.

VSIDS is not a static statistic. It takes the solving progress into consideration for branching. Experiments show that VSIDS is quite competitive compared with other branching heuristics on the number of branches needed to solve a problem. Because VSIDS is *variable-state independent* (i.e. scores are not dependent on the variable assignment), it is cheap to maintain. Experiments show that the decision procedure using VSIDS takes a very small percentage of the total run time even for problems with millions of variables.

More recently, the authors of SAT solver BerkMin [44] proposed another decision

with randomly generated instances.

[3]Learning will be explained in the next subsections.

scheme that pushes the idea of VSIDS further. Like VSIDS, the decision strategy is trying to decide on the variables that are "active recently". In VSIDS, the activity of a variable is captured by the score that is related to the literal's occurrences. In BerkMin, the authors propose to capture the activity by conflicts. More precisely, when a conflict occurs, all the literals in the clauses that are responsible for the conflict will have their scores increased. A clause is responsible for a conflict if it is involved in the resolution process of generating the learned clause (described in the following sections). In VSIDS, the focus on "recent" is captured by decaying the score periodically. In BerkMin, the scores are also decayed periodically. Moreover, the decision heuristic will limit the decision variable to be among the literals that occur in the last added clause that is unresolved. Their experiments indicate that the new decision scheme is more robust compared with VSIDS on the benchmarks tested.

In other efforts, the SAT solver satz [75] proposed the use of look-ahead heuristics for branching; and SAT solver cnfs [35] proposed the use of backbone-directed heuristics for branching. These branching heuristics share the common feature that they both seem to be quite effective on difficult random instances. However, these heuristics are also quite expensive compared with VSIDS and similar branching heuristics. Random SAT instances are usually much harder than structured instances of the same size. Current solvers can attack hard random 3-SAT instances with only several hundred variables. Therefore, the instances regarded as hard for random SAT are generally much smaller in size than the instances considered hard for structured SAT. Thus, while it may be practical to apply these expensive heuristics to the smaller random SAT instances, their overhead tends to be unacceptable for the larger structured SAT instances generated from real world applications.

2.3.3 Deduction Engine

The function `deduce()` serves the purpose of pruning the search space by "look ahead". When a branch variable is assigned a value, the entire clause database is simplified. The function `deduce()` needs to determine the consequences of the last decision and determine the needed variable assignments to make the instance satisfiable. It may return three status values. If the instance is satisfied under the current variable assignment, it will return `SATISFIABLE`; if the instance contains a conflicting clause, it will return `CONFLICT`; otherwise, it will return `UNKNOWN` and the solver will continue to branch.

There are various mechanisms with different deduction power and run time costs for the `deduce()` function. The correctness of the DLL algorithm will not be affected as long as the deduction rules incorporated are valid, i.e., it will not return `SATISFIABLE` when the instance contains a conflicting clause under the assignment. However, different deduction rules, or even different implementations of the same rule, can significantly affect the efficiency of the solver.

Over the years many different deduction mechanisms have been proposed. However, it seems that the unit clause rule [32] is the most efficient one because it requires relatively little computation but can prune relatively large search spaces. The unit clause rule states that for a certain clause, if all but one of its literals has been assigned the value 0, then the remaining (unassigned) literal must be assigned the value 1 for this clause to be satisfied, which is essential for the formula to be satisfied. Such clauses are called *unit clauses*, and the unassigned literal in a unit clause is called a *unit literal*. The iterative process of assigning the value 1 to all unit literals is called *unit propagation*, or sometimes called *Boolean Constraint Propagation (BCP)*. Almost all modern SAT solvers incorporate this rule in the deduction process. In a SAT solver, BCP usually takes the most significant part of the run time assuming the solver employs sufficiently efficient branching heuristics (which is the most common

case). Therefore, a SAT solver's efficiency is directly related to the implementation of its BCP engine. Various techniques have been developed to accelerate the speed of BCP. Some of them will be evaluated in the next chapter.

Besides the unit clause rule, there are other rules that can be incorporated into a deduction engine. This section will briefly review some of them. It should be pointed out that though many of the deduction mechanisms have been shown to work on certain classes of SAT instances, unlike the unit clause rule, none of them seems to work without deteriorating the overall performance of the SAT solver for general SAT instances.

One of the most widely known rules for deduction is the pure literal rule [32]. The pure literal rule states that if a variable only occurs in a single phase in all the unresolved clauses, then it can be assigned with a value such that the literal of the variable in that phase evaluates to 1. In practice, it is expensive to detect whether a variable satisfies the pure literal rule during the actual solving process. Therefore, most SAT solvers do not use pure literal rules in the deduction process by default.

Another often explored deduction mechanism is equivalence reasoning. Equivalence reasoning uses an additional data structure to capture the information that two variables are equivalent to each other (i.e., they must assume the same value to make the formula satisfiable). In particular, Li [74] incorporated equivalence reasoning into the satz [75] solver and observed that it is effective on some classes of benchmarks. In that work, equivalence reasoning is accomplished by a pattern-matching scheme for equivalence clauses. A related deduction mechanism using similar pattern matching schemes was proposed to include more patterns in the matching process for simplification purposes in deduction [77].

The unit literal rule basically guarantees that all the unit clauses are consistent with each other. This can be called 1-consistency. It is also possible to require that all the 2-literal clauses be consistent with each other, i.e. require 2-consistency, and

so on. Researchers have explored this idea [22, 131]. These approaches make two literal clauses consistent by maintain a transitive closure of all the 2-literal clauses. However, the overhead of maintaining this information seems to far outweigh any benefit gained from them on the average. Recently Bacchus [7, 8] proposed to include not only 2-literal clauses, but also some special clauses that are related to these 2-literal clauses into a resolution process called *hyper-resolution*. The experimental results show that hyper-resolution works much better than a transitive closure based algorithm and may dramatically reduce the number of nodes that need to be visited during the search. However, the applicability of this technique for large industrial instances is still unproven.

Recursive Learning [69] is a reasoning technique originally proposed in the context of learning with a logic circuit representation of a formula. Subsequent research [86] proposed to incorporate this technique in SAT solvers and found that it worked quite well for some benchmarks generated from combinational circuit equivalence checking problems.

2.3.4 Conflict Analysis

When a conflicting clause is encountered, the solver needs to backtrack and undo the decisions. Conflict analysis is the procedure that finds out the reason for a conflict and tries to resolve it. It tells the SAT solver that there exists no solution for the instance in a certain search space, and indicates a new search space to continue the searching.

The original DLL algorithm has the simplest conflict analysis method. For each decision variable, the solver keeps a flag indicating whether it has been tried in both phases (i.e. *flipped*) or not. When a conflict occurs, the conflict analysis procedure looks for the decision variable with the highest decision level that has not been flipped, marks it flipped, undoes all the assignments between that decision level and the

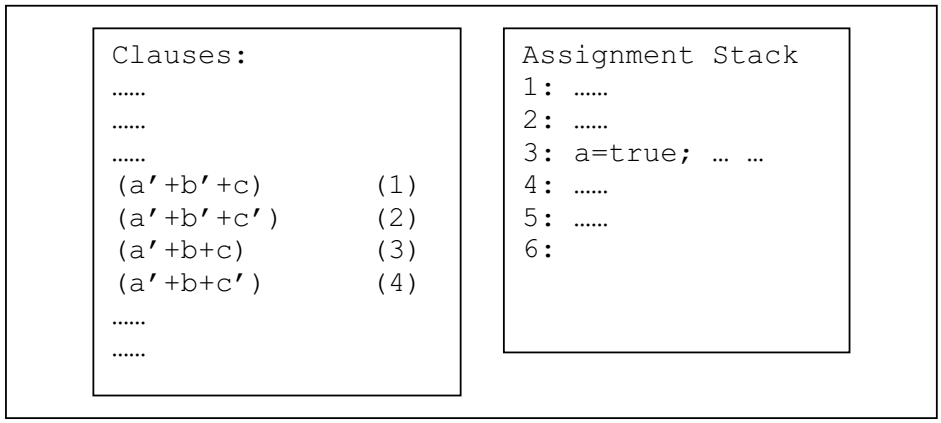


Figure 2.6: An example to illustrate non-chronological backtracking

current decision level, and then tries the other phase for the decision variable. This method is called *chronological backtracking* because it always tries to undo the last decision that is not flipped. Chronological backtracking works well for random SAT instances and is employed in some SAT solvers (e.g satz [75]). SAT solvers that employ chronological backtracking are actually equivalent to the recursive DLL algorithm (see Figure 2.2).

For structured instances generated from real world applications, chronological backtracking is often not effective in pruning the search space. More advanced conflict analysis engines can analyze the conflicting clauses encountered and get a better idea of the conflicts. Intelligent backtracking methods can determine the reason for the conflict and try to resolve it instead of blindly backtracking chronologically. This method can often backtrack to a smaller decision level than that of the last unflipped decision. Therefore, it is called *non-chronological backtracking*.

An example of non-chronological backtracking is given below. In Figure 2.6, four of the clauses in the formula are shown and numbered. The decision and assignment stack are shown to the right. It is assumed that none of the decision variables have been flipped yet. At decision levels 1 and 2, some variables are assigned but they do

not appear in the clauses of interest. At decision level 3, variable a is decided to be *true*. Some other variables are assigned at decision levels 4 and 5. Suppose at decision level 6, the solver decides to branch on b with the value *true*. Because of clauses 1 and 2, the formula is conflicting. Similarly, flipping b such that b is *false* will still make the formula conflicting because of clauses 3 and 4. Now the decision variable at decision level 6 (i.e. variable b) has been tried with both values 1 and 0, so the solver needs to backtrack. A chronological backtracking solver will try to flip the decision variable at decision level 5, because it is the decision variable with the highest decision level that has not been flipped. However, flipping variables at decision levels 4 and 5 can never resolve the conflict. The solver actually needs to flip a, which is assigned at decision level 3, in order to resolve the conflict. A non-chronological backtracking solver can correctly identify that in order to resolve this particular conflict, the solver needs to flip a. Therefore, it will backtrack to decision level 3 directly and skip decision levels 4 and 5.

During the conflict analysis process, information about the current conflict may be recorded as clauses and added to the original database. The added clauses, though redundant in the sense that they will not change the satisfiability of the original formula, can often help to prune search space in the future. This mechanism is called *conflict-driven learning*. Such learned clauses are called *conflict clauses* as opposed to conflicting clauses, which refer to clauses that generate conflicts. Essentially, conflict clauses capture the information that certain combinations of literals lead to conflict. Therefore, in future search, the solver can avoid the conflicts by avoiding the spaces covered by the conflict clauses.

Non-chronological backtracking, sometimes referred to as *conflict-directed backjumping*, was proposed first in the Constraint Satisfaction Problem (CSP) domain (e.g. [105]). This, together with conflict-driven learning, was first incorporated into SAT solvers by Silva and Sakallah in GRASP [88], and by Bayardo and Schrag in

relsat [64]. These techniques are essential for efficient solving of structured problems. Many modern SAT solvers such as SATO [136], chaff [95] and BerkMin [44] have incorporated similar techniques in the solving process. In the following the techniques used in modern SAT solvers for conflict analysis will be described in more detail.

Currently two approaches are used to illustrate the idea of conflict analysis; one uses an implication graph, and the other uses resolution. The implication graph based analysis is described here. The alternative formulation based on resolution will be described in Chapter 4.

The implication relationships of variable assignments during the SAT solving process can be expressed in an *implication graph*. A typical implication graph is illustrated in Figure 2.7. An implication graph is a directed acyclic graph (DAG). Each vertex represents a variable's assignment. A positive variable means it is assigned the value 1; a negative variable means it is assigned the value 0. The incident edges of each vertex are the reasons that lead to the variable's assignment. For example, in Figure 2.7, the incident edges to node $-V_{10}$ are from V_4 and $-V_2$, which means that if V_4 is *true* and V_2 is *false*, then V_{10} must be *false*. The vertices that have direct edges to a certain vertex v will be called v's antecedents. A decision vertex has no incident edge. The variable's decision level is denoted in the graph as the number within parenthesis.

In an implication graph with no conflict there is at most one vertex for each variable. A conflict occurs when the implication graph contains vertices for both 0 and 1 assignment of a variable. Such a variable is referred to as the *conflicting variable*. In Figure 2.7, the variable V_{18} is the conflicting variable.

In future discussion, when referring to an implication graph, only the connected component that has the conflicting variable in it will be examined. The rest of the implication graph is not relevant for the conflict analysis, and is therefore not considered in the discussion.

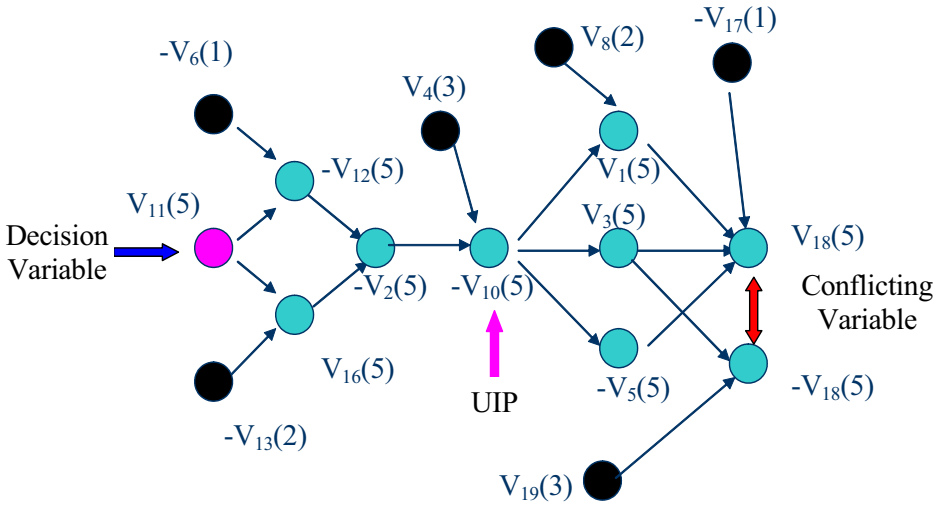


Figure 2.7: A typical implication graph. The conflict clause involved in the graph is $(V_3' + V_{19}' + V_{18}')$.

In an implication graph, vertex a is said to dominate vertex b if and only if any path from the decision variable of the decision level of a to the vertex b needs to go through a. A *Unique Implication Point (UIP)* [88] is a vertex at the current decision level that dominates both vertices corresponding to the conflicting variable. For example, in Figure 2.7, in the sub-graph of current decision level 5, V_{10} dominates both vertices V_{18} and $-V_{18}$, and therefore it is a UIP. The decision variable at the current decision level is always a UIP. Note that there may be more than one UIP for a certain conflict. In the example, there are three UIPs, namely, V_{11}, V_2 and V_{10}. Intuitively, a UIP is the single reason at the current decision level that implies the conflict. The UIPs are ordered starting from the conflict. In the previous example, V_{10} is the first UIP.

In actual implementations of SAT solvers, the implication graph is maintained by associating each assigned non-decision (i.e. implied) variable with a pointer to its *antecedent clause*. The antecedent clause of a non-decision variable is the unit clause that forced the variable to assume a certain value. By following the antecedent

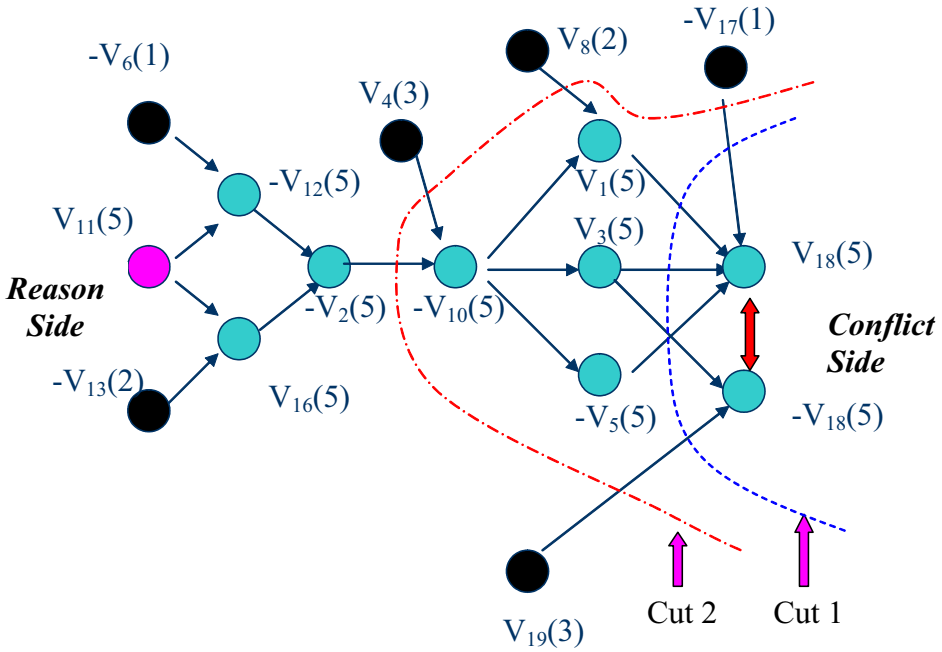


Figure 2.8: Cuts in an implication graph correspond to conflict clauses

pointers, the implication graph can be constructed when needed.

In SAT solvers, learning and non-chronological backtracking can be achieved by analyzing the implication graph. For example, in Figure 2.8, by examining the vertices that have outgoing edges crossing the line marked as "cut 1", it is easy to see that assignment $\{V_1, V_3, V_5', V_{17}', V_{19}\}$ will lead to V_{18} assigned with both value 1 and 0, and therefore it leads to conflict:

$$V_1 V_3 V_5' V_{17}' V_{19} \Rightarrow CONFLICT$$

The contrapositive is:

$$True \Rightarrow (V_1 V_3 V_5' V_{17}' V_{19})'$$

By De Morgan's law:

$$True \Rightarrow (V_1' + V_3' + V_5 + V_{17} + V_{19}')$$

Therefore, the clause $(V_1' + V_3' + V_5 + V_{17} + V_{19}')$ is redundant in the formula and can be added to the original database.

The example shows that a conflict clause can be generated by examining a bipartition of the implication graph. The partition has all the decision variables on one side (the *reason side*), and the conflicting variable in the other side (the *conflict side*). All the vertices on the reason side that have at least one edge to the conflict side comprise the reason for the conflict. Such a bipartition will be called a *cut*. Different cuts correspond to different learned clauses. For example, in Figure 2.8, clause $(V_1' + V_3' + V_5 + V_{17} + V_{19}')$ can be added as a conflict clause, which corresponds to cut 1. Similarly, cut 2 corresponds to clause $(V_2 + V_4' + V_8' + V_{17} + V_{19}')$.

By adding a conflict clause, the reason for the conflict is stated. If a cut is chosen such that the resulting clause contains only one variable that is assigned at the current decision level and all the remaining variables are assigned at decision levels lower than the current, then the resulting conflict clause is *asserting*. After backtracking, the clause will become a unit clause, and the unit literal will need to be assigned the value opposite to its current one. The backtracking level is the second highest decision level for all the literals in the clause (the highest level is current decision level). This will effectively result in a backtrack that can resolve the current conflict.

It is always desirable for a conflict clause to be an asserting clause. The unit variable in the asserting clause will be forced to assume a value and take the search to a new space to resolve the current conflict. To make a conflict clause an asserting clause, the partition can only have one variable at the current decision level with edges crossing the cut. This is equivalent to having one UIP at the current decision level on the reason side, and all vertices assigned after this UIP on the conflict side. After backtracking, the UIP vertex will become a unit literal, and make the clause an asserting clause.

Whenever a conflict occurs, the SAT solver will construct the implication graph

and try to find a cut that makes the resulting clause asserting. Such a cut can always be found because the decision variable at the current decision level is always a UIP. Different cuts represent different learning schemes. Some different learning schemes will be quantitatively evaluated in the next chapter. Learned clauses can be deleted later in the search process to control the database size. Several different clause deletion strategies have been proposed in literature [64, 44]. Currently, there is no clear consensus on the best clause deletion strategies. Thus this will not be discussed in this thesis.

2.3.5 Data Structure for Storing the Clause Database

Modern SAT solvers often need to solve instances that are quite large in size. Some instances generated from circuit verification problems may contain millions of variables and clauses. Moreover, during the SAT solving process, learned clauses are generated for each conflict encountered and may further increase the data-set size. Therefore, efficient data structures for storing the clauses are needed.

Most commonly, clauses are stored in a linear way, sometimes called a *sparse matrix representation*. The data-set can be regarded as a sparse matrix with variables as columns and clauses as rows. In a sparse matrix representation, the literals of each clause occupy their own space and no overlap exists between clauses. Therefore, the data-set size of a sparse matrix representation is linear in the number of literals in the clause database.

Earlier SAT solvers (e.g. GRASP [88] , relsat [64]) use pointer heavy data structures such as linked lists and array of pointers pointing to structures to store the clause database. Pointer heavy data structures, though convenient for manipulating the clause database (e.g. adding and deleting clauses), are not memory efficient and usually result in a high cache misses rate during the solving process because of lack of access locality. Newer SAT solvers often use more compact representation for the

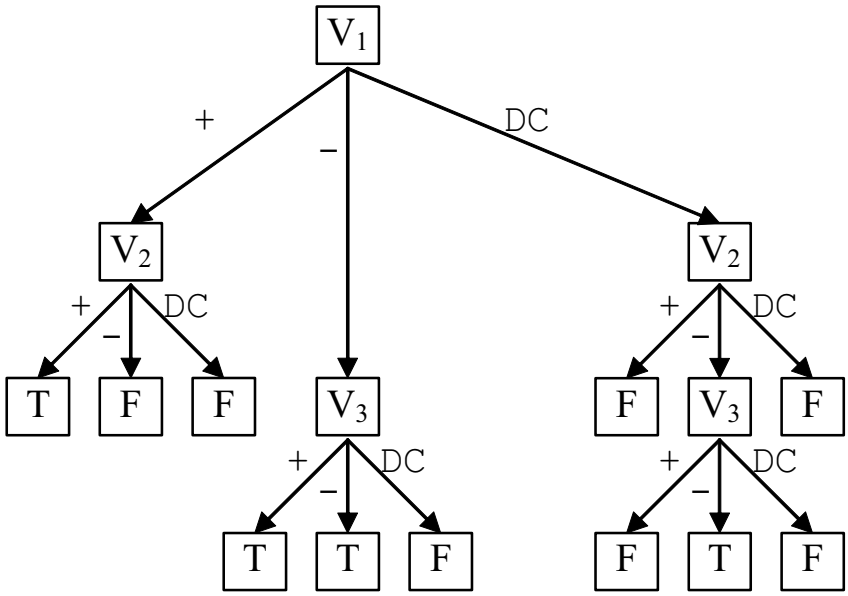


Figure 2.9: A trie representing the formula $(V_1 + V_2)(V_1' + V_3)(V_1' + V_3')(V_2' + V_3')$
.

clause database. For example, the SAT solver chaff [95] stores clause data in a single large array. Because arrays are not as flexible as linked lists, some additional garbage collection code is needed when clauses are deleted. The advantage of the array data structure is that it is very efficient in memory utilization. Moreover, because an array occupies contiguous memory space, access locality is increased. Experiments show that the array data structure has a big advantage over linked lists in terms of cache miss rates, and that translates to a substantial speed-up.

Researchers have proposed schemes other than sparse matrix representation for storing clauses. The authors of the solver SATO [137] proposed the use of a data structure called *trie* to store clauses. A trie is a ternary tree. Each internal node in the trie structure is a variable index, and its three children edges are labeled `Pos`, `Neg`, and `DC`, for positive, negative, and don't care, respectively. A leaf node in a trie is either *True* or *False*. Each path from root of the trie to a *True* leaf represents a clause. A trie is said to be ordered if for every internal node V, `Parent(V)` has a

smaller variable index than the index of variable V. The ordered trie structure has the nice property of being able to detect duplicate and tail subsumed clauses quickly. A clause is said to be *tail subsumed* by another clause if its first portion of the literals (a prefix) is also a clause in the clause database. For example, clause $(a+b+c)$ is tail subsumed by clause $(a+b)$. Figure 2.9 shows a simple clause database represented in a trie structure.

An ordered trie has obvious similarities with Binary Decision Diagrams. This has naturally led to the exploration of decision diagram style set representations. Chatalic *et al.* [24] and Aloul *et al.* [3] experimented with using Zero-suppressed Binary Decision Diagrams (ZBDDs) [93] to represent the clause database. A ZBDD representation of the clause database can detect not only tail subsumption but also *head subsumption*. Head subsumption is defined similar to tail subsumption except that the suffix of the subsumed clause is a clause in the database. Both works report significant compression of the clause database for certain classes of instances.

Based on current experimental data it does not seem that the data compression advantages of the trie and ZBDD are sufficient to justify the additional maintenance overhead of these data structures compared to the sparse matrix representation.

2.3.6 Preprocessing, Restart and other techniques

Preprocessing aims to simplify the instances before the regular solving begins in order to speed up the solving process. Usually the preprocessor of a SAT solver is just an extra deduction mechanism applied at the beginning of the search. Because the preprocessor will only be applied once in the solving process, it is usually possible to incorporate some deduction rules that are too expensive to be applied at every node of the search tree (e.g. pure literal rule [32], recursive learning [69], algebraic simplification [85]). The preprocessor can be applied on-line (within the solver) or off-line (it produces an equivalent instance to be fed to a solver). An overview of some

of the existing preprocessing techniques seem to indicate that the results of applying simplification techniques before the regular search are actually mixed [79].

When solving a SAT instance, a SAT solver sometimes may fall into a search space that is very hard to get out of because of bad decisions made early in the search process. To cope with this problem, a technique called random restart [46] is proposed. Random restart randomly throws away the current variable assignment and starts search from scratch. This technique is applied in modern SAT solvers such as chaff [95] and BerkMin [44]. In these solvers, when restart is invoked, even though the current search tree is abandoned, the solvers still keep some of the learned clauses from the past. Because of this, the previous search effort is not totally lost. Empirical data confirm that random restarts can increase the robustness of the SAT solvers.

Researchers have been extending the randomization idea of random restart to other aspects of the SAT solving process as well. For example, portfolio design [60] aims at using different solving strategies during one solving process in order to make the solver robust. Some researchers (e.g. [78, 101]) also propose to randomize backtracking. All in all, it seems that randomization is important for solving hard SAT instances.

2.4 Chapter Summary

This chapter gives an overview of the techniques used in SAT solving. Some basic concepts commonly used in SAT are introduced. Different techniques for SAT solving such as BDD, the DP algorithm, DLL search, Stålmarck's algorithm and stochastic methods are described. In particular, the most widely used SAT solving algorithm, namely the Davis Logemann Loveland (DLL) algorithm, is discussed in detail.

The DLL algorithm is the basis for all discussions in the rest of the thesis. Chapter 3 will quantitatively analyze some of the most widely used techniques involved in

implementing the DLL algorithm described in this chapter. Chapter 4 shows how to provide several additional features besides satisfiability checking for a SAT solver based on the DLL search. Chapter 5 will discuss how to apply DLL search for quantified Boolean formula evaluation.

Chapter 3

Efficient Implementation of the DLL Algorithm

Search algorithms based on the Davis Logemann Loveland (DLL) [31] algorithm are currently the predominant solutions for SAT. Over the years, researchers have proposed many heuristics to speed up the search and make SAT solvers more efficient on a wide variety of instances. As SAT solvers gain increased use in production environments, it is very important to evaluate the heuristics in a uniform framework in order to select the best heuristics for the best implementation of a general purpose SAT solver. This chapter quantitatively evaluates some of the major aspects of the implementation issues for a DLL based SAT solver.

The top level algorithm for DLL search was described in the previous chapter. In Figure 2.5, there are five main components of a SAT solving algorithm based on DLL search. Efficient (and effective) implementation of these functions is very important to the success of the overall SAT solving algorithm. SAT solvers differentiate themselves mainly on the implementations details and heuristic choices for these five functions.

- `preprocess()` is the function for preprocessing. It usually employs some deduction procedures to simplify the Boolean formula before the main loop begins. Many preprocessing techniques have been proposed in the literature (e.g.

[72, 85]) and comparative evaluations on the various techniques have been performed. One comparative result seems to suggest that the effectiveness of preprocessing is mixed: even though the instances may be simplified after the preprocessing, the run times to solve the simplified instances may not be reduced [77]. Because the preprocessor is only invoked once for each formula, usually efficient implementation is not an issue, while the algorithms and strategies employed are more important. This chapter will not deal with the efficient implementation of `preprocess()`, though some of the results obtained from this chapter (e.g. low cache miss rate data structure) may benefit the preprocessor.

- `decide_next_branch()` performs branching for the SAT solver. The branching heuristic is of great importance to the search process and is perhaps the most examined aspect of the SAT solving procedure (e.g. [20, 37, 61, 84, 95, 44]). Not surprisingly, comparative studies have also been carried out by various authors (e.g. [57, 84]). Unfortunately, there is little insight on why a certain branching heuristic works. Even though some of the heuristics are known to work better than others on certain kinds of SAT instances, the conclusion is often obtained by experimental evaluation. Different branching heuristics have very different implementation requirements and it is very hard to predict what kind of heuristics researchers may come up with in the future. Therefore, this chapter will not deal with the efficient implementation of the branching heuristics either.

- `deduce()` and `backtrack()` are two tightly related functions. The function `deduce()` performs reasoning on the variable assignments. In current SAT solvers, it essentially implements Boolean Constraint Propagation (BCP). The function `backtrack()` is the reverse of the BCP operation. It unassigns the variable values that have been assigned during the BCP process. The function `backtrack()` highly depends on the algorithms employed in the function

`deduce()`. These two functions usually take the most significant part of the run time for most SAT solvers, and are key to the overall efficiency of SAT solvers based on the DLL algorithm. BCP and backtracking will be examined in Section 3.2.

- `analyze_conflict()` is called when the solver finds a conflict in the database. This function determines the decision level to backtrack to, thus directing the search to a new space. It also performs conflict-driven learning during the conflict analysis process. Conflict-driven learning and non-chronological backtracking are very important for solving SAT instances generated from real world applications. In Section 3.3, a number of different schemes for conflict analysis are described, and their effects on the efficiency of the SAT solving process are evaluated.

The chapter is organized as follows. As this research is mainly focused on experimental evaluation of SAT algorithms on instances generated from real world applications, Section 3.1 describes the benchmarks used for the evaluation. Section 3.2 analyzes the various BCP algorithms for unit implication. Section 3.3 evaluates different learning and backtracking schemes. Careful design and implementation of data structure and algorithms can have significant impact on the memory behavior of the SAT solvers. Section 3.4 presents the analysis on cache performance of SAT solvers as a case study for efficient implementation. The chapter is summarized in Section 3.5.

3.1 Benchmark Description

Even though there are theoretical results on SAT solving algorithms (e.g. [55]), currently, the most effective and convincing method of evaluating a SAT algorithm is still based on empirically solving a given set of SAT instances. In this section, the

instances that will be used to evaluate the proposed SAT algorithms are described.

SAT as shown in [25], is a NP-Complete problem. Therefore, unless $P = NP$, there will always exist some instances that will take exponential run time in the size of the formula for any SAT solver. Some of the hard examples for DLL based SAT solvers are well known [129]. However, in practice, many of the real world SAT instances encountered can often be solved in reasonable time. There has been some discussions on the reasons for this phenomenon. For example, some researchers [9, 104] suggest that the SAT instances from real world often demonstrate a localized connectivity for the variables. More precisely, the "cut width" of the instances generated from practical applications are usually small. These discussions, while theoretically interesting, are usually not applicable in practice to determine *a priori* the hardness of a given SAT instance.

Traditionally, the DIMACS SAT benchmark [1] is regarded as a challenging test suite. The suite contains instances that are randomly generated and obtained from various practical applications. However, due to significant advances in SAT solving in recent years, most of the benchmarks in this suite can be solved in less than a couple of seconds by current state-of-the-art solvers, thus essentially rendering it useless for evaluating current SAT solvers.

Many methods for generating hard random SAT instances have been proposed before [26]. For example, randomly generated 3-SAT problems are known to be very hard when the clause to variable ratio is around 4.3. However, the kind of benchmarks that are synthesized or generated by random methods are usually quite unlike SAT instances encountered in the practical problems[1]. Many heuristics that are very effective on structured instances, such as learning, are known to be ineffective for random instances. In this thesis, only a few randomly generated instances will be used for evaluation.

[1]These instances are often called the *structured* instances, in contrast with the *random* instances.

As the research effort of this thesis is mainly driven by the demand from the Electronic Design Automation (EDA) area, the benchmarks used in the evaluation mainly come from EDA applications. These include benchmarks generated from microprocessor formal verification, bounded model checking, combinational equivalence checking and FPGA routing. To increase the variety of the benchmark selection, some SAT instances generated from AI planning are also included. These applications are briefly described now.

1. **Microprocessor Formal Verification**

   Verifying that a pipelined microprocessor is consistent with its non-pipelined specification can be accomplished utilizing uninterpreted functions as proposed by Burch *et al.* [19]. Various engines can be used to accomplish the reasoning task. Among them, SAT solvers have been shown to be the most efficient. These suites of benchmarks are generated by Velev [132]. They include verification of pipelined DLX microprocessors of varying complexity, and verification of a VLIW microprocessor with an architecture similar to the Intel Itanium processor.

2. **Bounded Model Checking**

   Bounded model checking was proposed by Biere *et al.* [11]. It performs model checking by unrolling the sequential circuit a finite number of clock cycles and uses a SAT solver (instead of BDDs, as regular model checkers do) to perform the reasoning on the unrolled circuit. Two suites of benchmarks obtained from bounded model checking are used in the evaluation. One is generated on some hand-made special circuits [10], the other is generated from real designs in IBM [126].

3. **Combinational Equivalence Checking**

   Combinational equivalence checking can be formed into a satisfiability problem

and solved by SAT solvers. It is accomplished by placing a miter (xor gate) at the outputs of the two circuits and checking if the output can ever be 1. The benchmarks used in this thesis are produced from checking the equivalence of each of the ISCAS-85 circuit with itself and also with a circuit that has gone through optimization.

4. **FPGA Routing**

   FPGA routing can be formulated as a SAT problem as shown by Nam *et al.* [97]. The method use clauses to represent constraints corresponding to connectivity and capacity of the routing channels. These benchmarks corresponds to the detailed routing of some industrial circuits.

5. **SAT Planning**

   Planning can be formulated as a SAT problem [65]. The instances are obtained from SAT-encoded problems such as "block worlds" and logistic planning.

Most of the benchmarks used are non-trivial to solve and are fairly large in size, ranging from less than one thousand variables to several hundred thousand variables in size. They are fairly representative of the problems usually encountered in real world applications. However, it should be pointed out that there are no "representative" SAT instances or a definitive test suite for SAT solving algorithms. Different SAT solvers have different strengths and weaknesses for various benchmarks.

3.2 Boolean Constraint Propagation

Unit clause propagation, sometimes called *Boolean Constraint Propagation(BCP)* is the core operation for deduction in a SAT solver[2]. The function of the BCP engine is to find all the unit clauses and assign the unit literals with proper values. The BCP

[2]Part of the work described in this section was first reported in SAT2003 [142].

engine does this iteratively until either no unit clause is left in the clause database, or a conflicting clause exists. In the first case, the solver will make another decision and then continue the deduction; in the second case, the solver will invoke the conflict analysis procedure and backtrack. Since BCP usually takes the most time in SAT solving, the BCP engine is the most important part of a SAT solver and usually dictates the data structure and organization of the solver.

One aspect of the SAT solving process that is tightly related to BCP is backtracking. The backtracking procedure undoes the variable assignments performed by the BCP process to restore the SAT solver to a previous state. Since these two procedures are essentially the complement of each other, in the rest of the thesis they will be considered as one integrated part and evaluated as a unit. When BCP mechanisms or BCP engines are mentioned in the thesis, the respective backtracking schemes are referred to as well.

In Table 3.1 the statistics such as number of clauses and literals of the instances that will be used in this section are listed. The instances come from bounded model checking (longmult12, barrel9, ibm), FPGA routing (too_large, alu, vdafs), microprocessor verification (2dlx, 9vliw, 5pipe) and SAT planning (bw_large).

Table 3.2 shows the run time statistics of zchaff [138], a state-of-the-art SAT solver, on the SAT instances listed in Table 3.1. The solver is compiled using g++ 2.95 with the compiler option `-O3 -pg`. The `-pg` option makes it possible to use profiler `gprof` to determine the percentage of run time spent on each function. The profiler `gprof` uses sampling techniques to obtain the run time spent in each functions[3]. The table also contains the total run time and the run time spent on BCP as well as its percentage of the total time for each instance. The experiments were carried out on a Pentium III 1.13G machine with 1G memory.

[3]The statistics provided by `gprof` may not be accurate in reflecting the cache behavior of the profiled program because `gprof` needs to allocate and access memory for profiling purposes. Therefore, statistics shown here can only be regarded as a rough estimate.

Instance Name	**Num. Variable**	**Num. Clauses**	**Num. Literals**	**Lits/ Per Cl.**	**Sat Unsat**
alu2fs3w8v262	2919	49336	263424	5.34	SAT
2dlx_ca_mc_ex_bp_f	3250	24640	69012	2.80	UNSAT
bmc-galileo-9	63624	326999	832623	2.55	SAT
too_largefs3w9v262	3928	95285	677662	7.11	SAT
too_largefs3w8v262	2946	50416	271056	5.38	UNSAT
bw_large.d	5886	122412	272739	2.23	UNSAT
c5315	5399	15024	34628	2.30	UNSAT
vdafs3w10v262	6412	195838	1419596	7.25	SAT
c7552	7652	20423	46377	2.27	UNSAT
bmc-ibm-12	39598	194660	514254	2.64	SAT
5pipe_5_ooo	10113	240892	705754	2.93	UNSAT
longmult12.dimacs	5974	18645	44907	2.41	UNSAT
9vliw_bp_mc	20093	179492	507734	2.83	UNSAT
barrel9.dimacs	8903	36606	102370	2.80	UNSAT

Table 3.1: SAT instances used in this section

Instance Name	**Total Runtime (s)**	**BCP Time (s)**	**BCP Percentage**
alu2fs3w8v262	3.53	4.65	75.89%
2dlx_ca_mc_ex_bp_f	1.59	2.16	73.64%
bmc-galileo-9	10.72	14.85	72.20%
too_largefs3w9v262	29.11	36.88	78.93%
too_largefs3w8v262	35.17	40.50	86.84%
bw_large.d	5.18	6.41	80.85%
c5315	18.15	22.93	79.14%
vdafs3w10v262	204.25	261.47	78.12%
c7552	52.48	65.71	79.87%
bmc-ibm-12	39.05	49.07	79.59%
longmult12.dimacs	261.46	300.55	86.99%
9vliw_bp_mc	253.88	373.02	68.06%

Table 3.2: BCP time as a percentage of total runtime for zchaff

The table shows that for most of the instances, zchaff spends around 80% of the total run time in BCP. It should be pointed out that zchaff has one of the most efficient implementations of BCP among currently available SAT solvers. Other SAT solvers may spend an even higher percentage of run time on BCP. Therefore, it is very important to make BCP as efficient as possible in order to improve the overall efficiency of the SAT solver.

3.2.1 Counter Based BCP

A simple and intuitive implementation for BCP is to keep counters for each clause. This scheme is attributed to Crawford and Auton (see [28] and [135]). Similar schemes have been subsequently employed in many solvers such as GRASP [88], relsat [64], satz [75] etc.

Counter-based BCP has several variations. The simplest counter-based algorithm can be found in solvers such as GRASP [88]. In this method, each clause keeps two counters, one for the number of value 1 literals in the clause and the other for the number of value 0 literals in the clause. It also keep the information about the total number of literals in this clause. Each variable has two lists that contain all the clauses where that variable appears as a positive and negative literal respectively. When a variable is assigned a value, all of the clauses that contain the literals corresponding to this variable will have their counters updated. The data structure of such an implementation is as follows:

```
struct Clause {
    unsigned * literal_array;
    unsigned num_total_literal;
    unsigned num_value_1_literal_counter;
    unsigned num_value_0_literal_counter;
```

```
};
struct Variable {
    unsigned value;
    Clause * pos_occurrence;
    Clause * neg_occurrence;
};
struct ClauseDatabase {
    unsigned num_clauses;
    Clause * clause_array;
    unsigned num_variables;
    Variable * variable_array;
};
```

Since each clause contains two counters in this BCP scheme, it will be referred to as the *2-Counter Scheme*. After a variable assignment, there are three cases for a clause with an updated counter. If the clause's value 0 count equals the total number of literals in the clause, then it is a conflicting clause. If the clause's value 0 count is one less than the total number of literals in the clause and the value 1 count is 0, then the clause is a unit clause. Otherwise, the clause is neither conflicting nor unit, nothing needs to be done for the clause. If a clause is determined to be a unit clause, the solver has to traverse the array of the literals in that clause to find the free (unassigned) literal to imply. Given a Boolean formula that has m clauses and n variables with l literals for each clause on average, then in the formula each variable occurs lm/n times on average. Using this BCP mechanism, whenever a variable gets assigned, on the average lm/n counters need to be updated.

An alternative implementation can use only one counter for each clause. The counter would be the number of non-zero literals in the clause. The data structure for the clause would then be:

```
struct Clause {
    unsigned num_total_literals;
    unsigned * literal_array;
    unsigned num_value_non_0_literal_counter;
};
```

Notice that number of total literals in a clause is not changing during the search, so it is not a counter. The storage it occupies can even be saved by using some tricks to mark the end of the literal array. For example, it can be accomplished by setting the highest bit of the last element in the array to 1 to indicate the end of the array. Since each clause only contains one counter in this scheme, it will be referred to as the *1-Counter Scheme*.

In this scheme, when a variable is assigned a value, the solver only needs to update (reduce by 1) all the counters of clauses that contain the value 0 literal of this variable. There is no need to update clauses that contain the value 1 literal of this variable. After the assignment, if a clause's non-zero counter is equal to 0, then the clause is conflicting. If it is 1, the solver can traverse the literal array and find the one literal that is not 0. There are two cases: if it is a free (unassigned) literal, then the clause is a unit clause and the literal is implied; otherwise it must already evaluate to 1 and the clause is already satisfied.

This scheme essentially cuts the number of clauses that need to be updated by half. Whenever a variable gets assigned, on the average only $lm/2n$ counters need to be updated. However, it does need to traverse clauses more often than the first scheme because if a clause has one literal evaluating to 1 and all the rest evaluate to 0, the 2-Counter scheme will know that the clause is already satisfied and no traversal is necessary. But a solver using the 1-Counter scheme still needs to traverse the clause literals because it does not know whether the literal evaluates to 1 or is unassigned.

Notice that in the data structure described above, the counters associated with

each clause are stored within the clause structure. This is actually not good from a cache performance point of view. When a variable is assigned, many counters need to be updated. If the counters are located far from each other in memory, the update may cause a lot of cache misses. Therefore, to improve data locality, the counters should be located as close as possible with each other in memory. It is possible to allocate a separate continuous memory space to store the counters. In that case, the structure becomes:

```
struct Clause {
    unsigned * literal_array;
    unsigned num_total_literal;
};
struct ClauseDatabase {
    unsigned num_clauses;
    Clause * clause_array;
    unsigned * clause_counter_array;
    unsigned num_variables;
    Variable * variable_array;
};
```

This alternative data structure to implement the 1-Counter Scheme will be called *Compact 1-Counter Scheme*.

Counter-based BCP engines are easy to understand and implement, but these schemes are not the most efficient ones. On backtracking from a conflict, the counters need to be updated for the variables unassigned during the backtracking. Each undo for a variable assignment will also update lm/n or $lm/2n$ counters on average, depending on the schemes used. Modern solvers usually incorporate learning mechanisms in the search process. As shown in Table 3.3 in Section 3.2.3, learned

clauses are usually much larger than the original clauses, each containing tens or even hundreds of literals. Therefore, the average clause length l is quite large during the search, thus making a counter-based BCP engine relatively slow.

3.2.2 Pointer Based BCP

The SAT solver SATO [135] uses a new mechanism for BCP without counters. In this mechanism, each clause has two pointers associated with it, called the head and tail pointer respectively. In SATO (as well as most other SAT solvers), a clause stores all its literals in an array. Initially, the head pointer points to the first literal of the clause (i.e. beginning of the array), and the tail pointer points to the last literal of the clause (i.e. end of the array). The solver maintains the invariant that for a given clause, all literals located before the head pointer are assigned the value 0, and all literals located after the tail pointer are assigned the value 0. Therefore, if a literal is not assigned the value 0, it must be located between the head and tail pointers.

In this BCP scheme, each variable keeps four linked lists that contain pointers to clauses. The linked lists for the variable v are `clause_of_pos_head(v)`, `clause_of_neg_head(v)`, `clause_of_pos_tail(v)` and `clause_of_neg_tail(v)`. Each of these lists contains the pointers to the clauses that have their head/tail literal in positive/negative phases of variable v. If v is assigned with the value 1, for each clause C in `clause_of_neg_head(v)`, the solver will search for a literal that does not evaluate to 1 from the head literal to the tail literal of C. Notice the head literal of C must be a literal of v in negative phase. During the search process, four cases may occur:

1. If during the search a value 1 literal is encountered first, then the clause is satisfied; nothing more needs to be done for this clause.

2. If during the search a free literal l is encountered that is not the tail literal, then

the solver removes C from `clause_of_neg_head(v)` and adds C to the head list of the variable of l. This operation is referred to as *moving the head literal* because in essence the head pointer is moved from its original position to the position of l.

3. If all literals in between these two pointers are assigned the value 0, but the tail literal is unassigned, then the clause is a unit clause, and the tail literal is the unit literal.

4. If all literals in between these two pointers and the tail literal are assigned the value 0, then the clause is a conflicting clause.

Figure 3.1 shows these four cases of the Head/Tail BCP scheme when the head pointer is moved (i.e. variable V_2 is assigned the value 0). Similar actions are performed for `clause_of_neg_tail(v)`, only the search is in the reverse direction (i.e. from tail to head). The clauses in `clause_of_pos_head(v)` and `clause_of_pos_tail(v)` will not be touched in this case. This BCP algorithm is referred to as the *Head/Tail list* algorithm.

The head/tail list method is an improvement over the counter-based schemes because when a variable is assigned value 1, the clauses that contain the positive literals of this variable are not visited, and vice-versa. As each clause has only two pointers, whenever a variable is assigned, on average the status of only m/n clauses needs to be updated assuming that head/tail literals are distributed evenly in either phase. Even though the work needed to be done for each update is different from the counter-based mechanism, in general, the head/tail mechanism is still faster.

For both the counter-based algorithm and the head/tail list-based algorithm, undoing a variable's assignment during backtrack has about the same computational complexity as assigning the variable. The authors of the solver chaff [95] proposed a BCP algorithm called *2-literal watching* that can avoid this problem. The algorithm

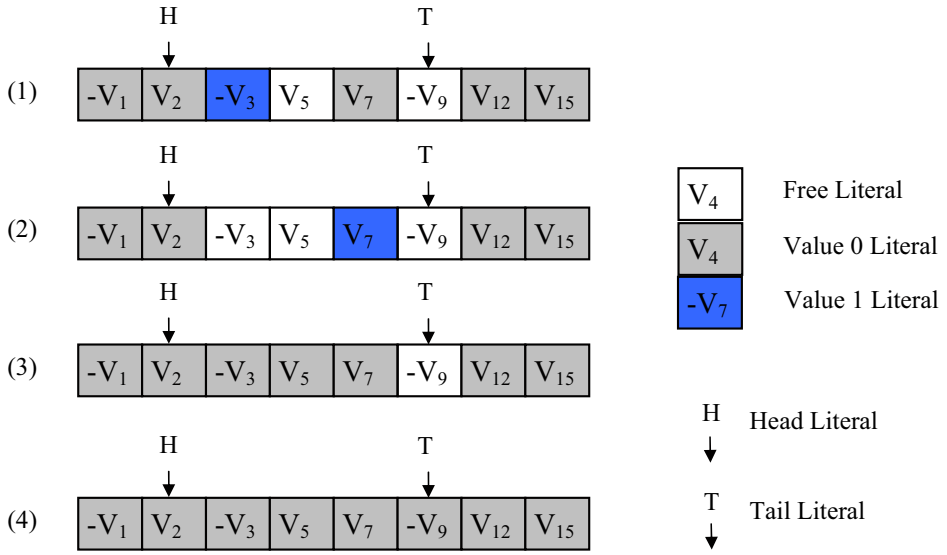


Figure 3.1: Four cases of Head/Tail BCP scheme

is based on the observation that for a given clause, as long as it contains two literals that are not assigned to value 0, then it will neither be unit nor conflicting. Therefore, the solver only needs to keep track of two non-zero valued literals for each clause in order to detect unit clauses and conflicting clauses in the clause database.

Similar to the head/tail list algorithm, 2-literal watching has two special literals for each clause called watched literals. Each variable has two lists containing pointers to all the watched literals corresponding to it for both phases. The lists for variable v is denoted as `pos_watched(v)` and `neg_watched(v)`. In contrast to the head/tail list scheme in SATO, there is no imposed order on the two pointers within a clause, and a pointer can move in either direction, i.e. the 2-literal watching scheme does not have to maintain the invariant of the head/tail scheme. Initially the watched literals are free (i.e. unassigned). When a variable v is assigned value 1, for each literal p pointed to by a pointer in the list of `neg_watched(v)` (notice p must be a literal of v with negative phase), the solver will search for a non-zero literal l in the clause containing p. There are four cases that may occur during the search:

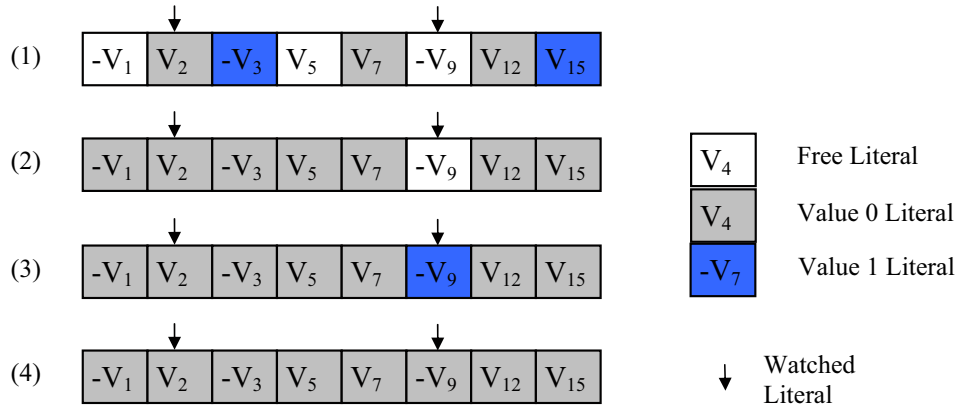


Figure 3.2: Four cases of 2-Literal Watching BCP scheme

1. If such a literal l exists and it is not the other watched literal, then the solver removes the pointer to p from `neg_watched(v)`, and adds the pointer to l to the watched list of the variable of l. This operation is referred as *moving the watched literal* because in essence one of the watched pointers is moved from its original position to the position of l.

2. If the only such l is the other watched literal and it is free, then the clause is a unit clause, with the other watched literal being the unit literal.

3. If the only such l is the other watched literal and it evaluates to 1, then nothing needs to be done for this clause.

4. If all literals in the clause is assigned the value 0 and no such l exists, then the clause is a conflicting clause.

Figure 3.2 shows these four cases when a watched pointer need to be moved (i.e. variable V_2 is assigned the value 0). Compared with the counter-based schemes, 2-literal watching has the same advantage as the head/tail list mechanism. Moreover, unlike the other mechanisms, undoing a variable assignment in the 2-literal watching scheme takes constant time. No action is required to update the pointers for the

literals being watched when backtracking take place. This is because the watched literals were the last literals in the clause to be set to 0, and thus any backtrack will ensure that the watched literals will no longer be 0. Therefore, it is significantly faster than both counter-based and head/tail list mechanisms for BCP. In Figure 3.3, a comparison of the 2-literal watching and the head/tail list mechanism is shown.

3.2.3 Experimental Results for Different BCP Schemes

In this section, different BCP schemes discussed in the previous sections are experimentally evaluated. The experiments are carried out by modifying the zchaff SAT solver [138]. In the original zchaff code, different BCP methods may lead the solver into different search paths since the variables may be implied in different order by different BCP mechanisms. To solve this problem, the code is modified such that the solver will follow the same search path regardless of the implication methods. This incurs a very small overhead that is negligible. Because of the modification, the search paths are not the same as those of the default zchaff solver, whose run times to solve the benchmarks are shown in Table 3.2.

Unfortunately, there seems to be no *reference implementation* of the Head/Tail list BCP method available in the public domain. The best SAT solver using this method (BerkMin [44]) is a closed-source solver and thus no implementation detail is available. The open source solver that implemented this BCP method (SATO [136]) is clearly inferior compared with the BerkMin's implementation as will be observed in Section 3.4. The implementation used here may not be as efficient as the implementation in BerkMin, therefore it may not be representative of the best performance that can be achieved by this BCP method. However, it is expected to be reasonably close.

Some of the common statistics for solving the benchmark instances using the controlled zchaff solver are shown in Table 3.3. The statistics shown include the

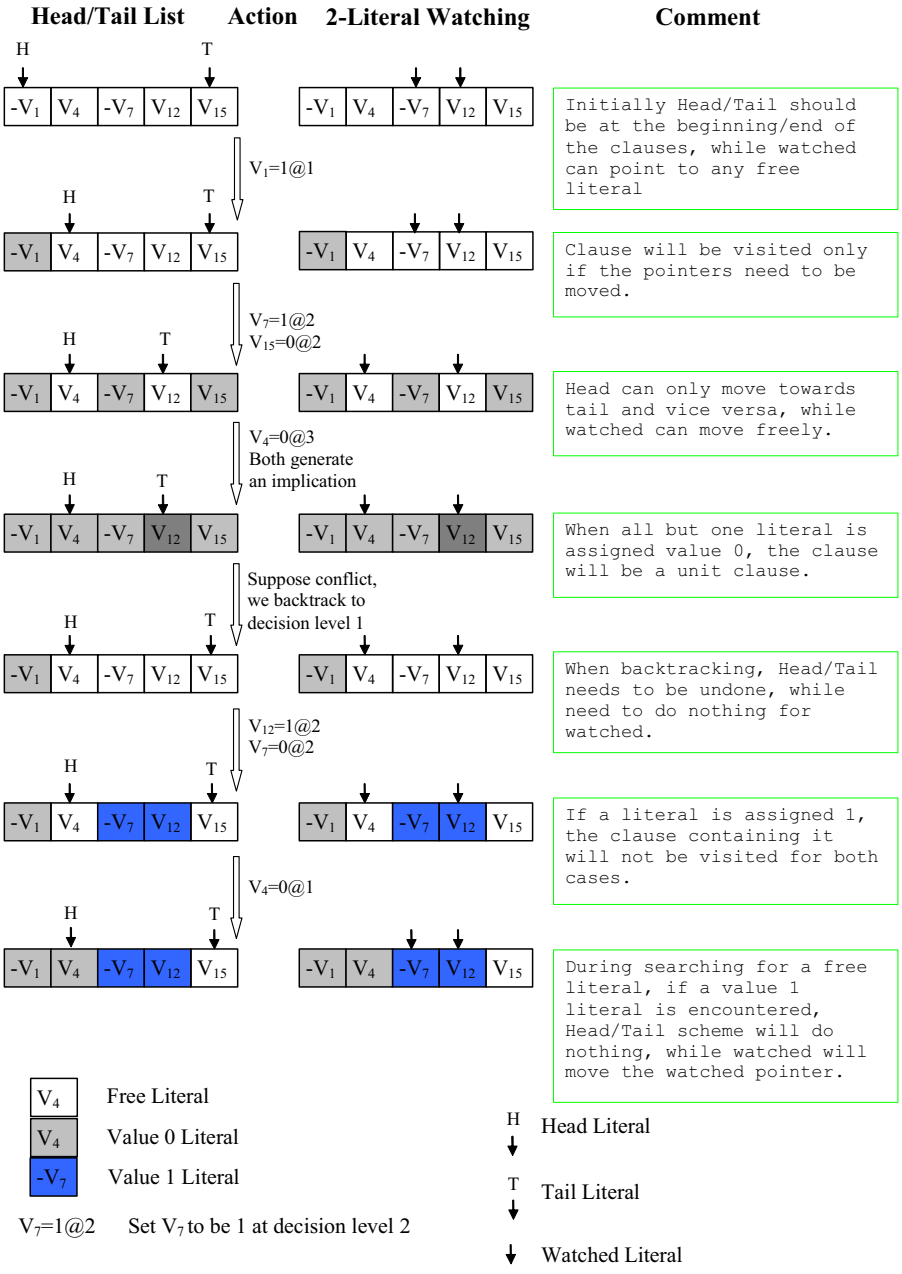


Figure 3.3: Comparison of 2-literal watching scheme and Head/Tail List scheme

Instance Name	**Num. Lrned Clauses (k)**	**Num. Lrned Literals (k)**	**Learned Lits/Cls**	**Num. Implications (M)**
alu2fs3w8v262	16.36	150.37	9.19	1.42
2dlx_ca_mc_ex_bp_f	8.05	590.62	73.41	1.51
bmc-galileo-9	2.37	80.50	33.94	4.63
too_largefs3w9v262	65.93	787.24	11.94	7.97
too_largefs3w8v262	71.77	2846.98	39.67	9.47
bw_large.d	14.90	398.58	26.75	11.11
c5315	83.00	7060.77	85.07	30.17
vdafs3w10v262	273.07	4180.59	15.31	50.92
c7552	192.84	24358.61	126.32	82.31
bmc-ibm-12	44.05	5952.62	135.12	82.34
longmult12	134.67	19232.97	142.82	126.06
9vliw_bp_mc	515.33	219494.70	425.93	147.39

Table 3.3: Instances for evaluating different BCP schemes

number of learned clauses and literals, as well as learned clause to literal ratio. The numbers of implications needed for solving these instances are also shown. Table 3.4 shows the total run time for solving these benchmarks. The experiments were carried out on a Dell PowerEdge 1500sc machine with a 1.13Ghz Pentium III CPU and 1G memory.

From the table it is easy to see that different implementations of the BCP mechanism have significant impact on the runtime of the SAT solver. Comparing the best performing mechanism, i.e. the 2-literal watching scheme, with the worst scheme, i.e. the 2-Counter scheme, for some benchmarks there is a speed difference of more than 10x (e.g. C7552, 9vliw). Comparing the 2-Counter Scheme and the 1-Counter Scheme, it seems that the latter is always faster. The compact 1-Counter scheme is faster than the regular 1-Counter scheme. The reason will be explained in Section 3.4. The Head/Tail BCP scheme does not perform as well as the 2-literal watching scheme for many of the benchmarks tested.

Instance Name	Counter Based			Pointer Based	
	2-Counter	**1-Counter**	**Compact 1-Cnt**	**Head/Tail**	**Watch**
alu2fs3w8v262	12.39	8.90	6.35	13.77	4.43
2dlx_ca_mc_ex_bp_f	7.42	5.55	5.09	6.94	3.10
bmc-galileo-9	16.53	12.23	11.51	16.21	8.10
too_largefs3w9v262	196.11	130.46	71.71	161.67	36.77
too_largefs3w8v262	712.49	320.65	142.65	278.59	41.45
bw_large.d	118.46	49.06	39.94	67.72	25.54
c5315	420.81	267.47	173.02	227.27	66.48
vdafs3w10v262	3475.18	2299.45	1234.71	2239.60	352.88
c7552	3053.57	1795.27	873.77	1030.72	260.93
bmc-ibm-12	418.73	322.43	283.25	416.85	180.52
longmult12	3839.28	2169.79	984.40	1514.88	432.97
9vliw_bp_mc	8706.39	5217.37	2555.89	1431.54	747.37

Table 3.4: Runtime for solver with different BCP mechanisms

3.3 Learning Schemes

When a conflicting clause is encountered, the solver needs to backtrack and undo the decisions[4]. Conflict analysis is the procedure that finds the reason for a conflict and tries to resolve it. It tells the SAT solver that there exists no solution for the problem in a certain search space, and indicates a new search space to continue the searching.

Different learning schemes can affect the efficiency of the SAT solver greatly. A good learning scheme should reduce the number of decisions needed to solve certain problems as much as possible. The effectiveness of a learning scheme is very hard to determine *a priori*. Generally speaking, a shorter clause can constrain the search space more than a longer clause. Therefore, a common belief is that the shorter the learned clause, the more effective the learning scheme. However, SAT solving is a dynamic process. It has a complex interplay between the learning schemes and

[4]Part of the work described in this section was first reported in ICCAD 2001 [139].

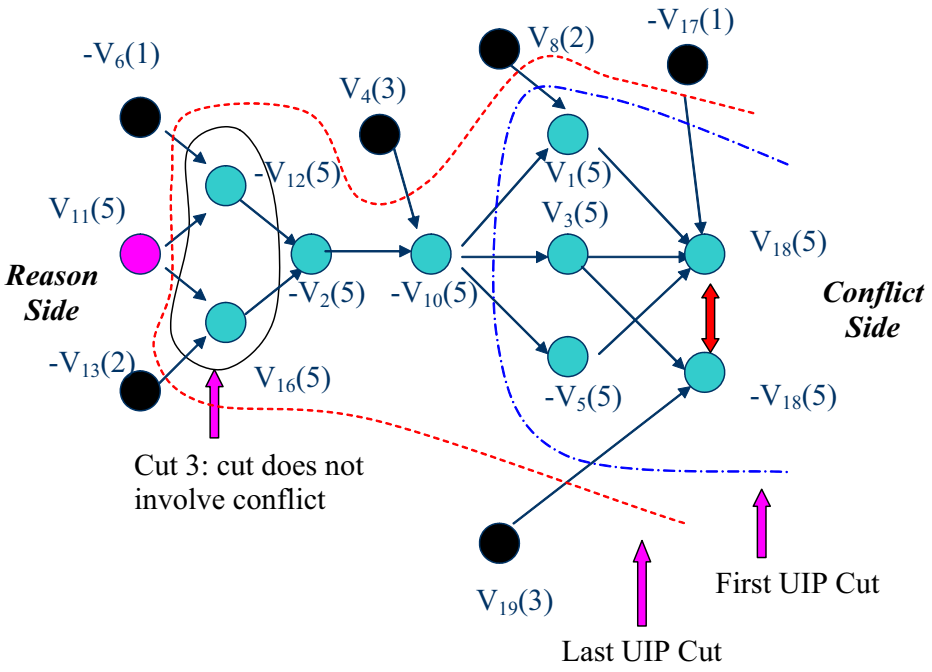


Figure 3.4: Different learning schemes

the searching process. Therefore, the effectiveness of learning schemes can only be determined by empirical data for the entire solution process.

As mentioned in Chapter 2, learning and non-chronological backtracking can be achieved by analyzing the implication graph. A conflict clause is generated by a bipartition of the implication graph. The partition has all the decision variables on the *reason side*, and the conflicting variable on the *conflict side*. All the vertices on the reason side that have at least one edge to the conflict side comprise the reason for the conflict. This bipartition is called a *cut*. Different cuts correspond to different learning schemes. In this section, some of the existing learning schemes and some newly proposed schemes are examined. The implication graph in Figure 3.4 will be used to illustrate the discussion.

3.3.1 Different Learning Schemes

Relsat [64] is one of the first SAT solvers to incorporate learning and non-chronological backtracking. The relsat conflict analysis engine generates the conflict clause by recursively resolving the conflicting clause with its antecedent, until the resolved clause includes only decision variables of the current decision level and variables assigned at decision levels smaller than the current level. In the implication graph representation, the relsat engine will put all variables assigned at the current decision level, except for the decision variable, on the conflict side; and put all the variables assigned prior to the current level and the current decision variable on the reason side. In the example, the conflict clause added would be:

$$(V'_{11} + V_6 + V_{13} + V'_4 + V'_8 + V_{17} + V'_{19})$$

This learning scheme will be called the *relsat scheme*.

A different learning scheme is implemented in GRASP [88]. GRASP's learning scheme is different from relsat's in the sense that it tries to learn as much as possible from a conflict. In GRASP, there are two kind of decision variables, a *real* one and a *fake* one. Real decision variables do not have antecedent edges in the implication graph, while fake ones do. If the current decision variable is a real decision variable, the GRASP learning engine will add each reconvergence between UIPs in the current decision level as a learned clause. In the example, if V_{11} is a real decision (i.e. it has no antecedent), when the conflict occurs, the GRASP engine will add one clause into the database corresponding to Cut 3 shown in Figure 3.4 . This corresponds to the clause

$$(V'_2 + V_6 + V'_{11} + V_{13})$$

Moreover, GRASP will also include a conflict clause that corresponds to the partition where all the variables assigned at the current decision level after the first UIP are put on the conflict side. The rest of the vertices are put on the reason side. This

corresponds to the First UIP cut as shown in figure 3.4. The clause added will be $(V_{10} + V_8' + V_{17} + V_{19}')$.

After backtracking, the conflict clause will be an asserting clause, which forces V_{10} to flip. Note that V_{10} was not a decision variable before. GRASP will regard V_{10} as a fake decision. The decision level of V_{10} remains unchanged, and it now has an antecedent clause. This essentially means that the solver is not done at the current decision level yet. So it only flips a variable (at the current decision level) but does not backtrack. This mode of the analysis engine will be called *flip mode*.

If the following deduction finds that flipping a variable still leads to a conflict, the GRASP conflict analysis engine will enter *backtracking mode*. Besides the clauses that have to be added in the flip mode, it also adds another clause called the *back clause*. The cut for the back clause will put all the vertices at the current decision level (including the fake decision variable) on the conflict side, and all other vertices on the reason side. For the example, suppose the decision variable V_{11} is actually a fake decision variable, with antecedent clause $(V_{21} + V_{20} + V_{11}')$ (i.e. there are actually two edges lead to V_{11}, from V_{20}' and V_{21}' respectively). Then, besides the two clauses added before, the GRASP engine will add another clause

$$(V_{21} + V_{20} + V_6 + V_{13} + V_4' + V_8' + V_{17} + V_{19}')$$

This clause is a conflicting clause, and it only involves variables assigned before the current decision level. The conflict analysis engine will continue to resolve this conflict, and bring the solver to an earlier decision level, possibly performing a non-chronological backtracking. In the following, this learning scheme will be called the *GRASP scheme*.

Besides these two learning schemes, many more options exist. One obvious learning scheme is to add only the decision variables involved in the conflict to the learned conflict clause. In the implication graph representation, this corresponds to making

the cut such that only the decision variables are on the reason side, and all the other variables are on the conflict side. This scheme will be called the *Decision scheme*. Note that only the decision variables *involved* in the conflict are put in the learned clause. The decision variables that are not in the connected component of the implication graph with conflict are not involved. It is no good to include *all* the decision variables of the current search tree in the conflict clause, because such a combination of decisions will never occur again in the future unless the solver restarts. Thus the learned clause will hardly help prune the search space at all.

The decision variables are the last UIPs for each decision level. A more careful study of the Decision scheme suggests making a partition not limited at decision variables, but after any UIPs of each of the decision levels. One of the choices is to make a partition after the first UIP of each decision level. This scheme will be called the *All-UIP scheme* because all of the literals in the conflict clause are UIPs. Because UIPs of a lower level depend on the partition, the solver need to make the partition start from the current decision level down to decision level 1. More precisely, the solver needs to find the first UIP of the current decision level, then all variables of current decision level assigned after it that have a path to the conflicting variable will be on the conflict side; the rest will be on the reason side. Then the solver will proceed to the decision level prior to current one, and so on, until reaching decision level 1.

It may also be desirable to make the conflict clause as relevant to the current conflict as possible. Therefore, the solver may want to make the partition closer to the conflicting variable. As mentioned earlier, to make the conflict clause an asserting clause, the solver needs to put one UIP of the current decision level on the reason side. Therefore, by putting all variables assigned after the *first* UIP of the current decision level that have paths to the conflicting variable on the conflict side, and everything else on the reason side, the solver can get a partition that is as close to the conflict

as possible. This will be called the *1-UIP* scheme. Notice that the relsat scheme is actually similar to the 1-UIP scheme except it puts the *last* UIP of the current decision variable in the conflict clause.

The 1-UIP scheme is different from the All-UIP scheme in that only the current decision level is required to have its first UIP on the reason side, just before the partition. It is also possible to require that the immediately previous decision level have its first UIP on the reason side just before the partition. This will make the conflict have only one variable that was assigned at the immediately previous decision level. This will be called the *2-UIP scheme*. Similarly, the solver can have *3-UIP scheme* etc., up to *All-UIP*, which makes the conflict clause have only one variable for any decision level involved. Notice the difference of All-UIP scheme with Decision scheme. In Decision scheme, the last UIP for each decision level is put into the conflict clause, while in All-UIP scheme, the first UIPs are put into the conflict clause.

All of the learning schemes discussed can be implemented by a traversal of the implication graph. The time complexity of the traversal is $O(V + E)$. Here V and E are the number of vertices and edges of the implication graph respectively.

3.3.2 Experimental Results

The learning schemes discussed are implemented in the SAT solver zchaff [138] for empirical evaluation. Zchaff utilizes the highly optimized chaff's BCP engine and VSIDS decision heuristic, thus making the evaluation more interesting because some large benchmarks can be evaluated in a reasonable amount of time.

Zchaff's default branching scheme VSIDS [95] depends on the added conflict clauses and literal counts for making branching decisions. Therefore, different learning schemes may also affect the decision strategy. Decision heuristics employed in most modern SAT solvers such as GRASP [88], chaff [95] and BerkMin [44] depend on the learned clauses. Therefore, the decision strategy of zchaff is representative of

this common trait. In order to eliminate the effect of learning on decision heuristic, a fixed branching strategy is also implemented, which branches on the first unassigned variable with the smallest predetermined variable index.

The test suite consists of benchmarks from bounded model checking, microprocessor formal verification and AI planning. The experiments are conducted on a dual Pentium III 833Mhz machines running Linux (only one cpu is used), with 2G bytes physical memory. Each CPU has 32k L1 cache and 256k L2 cache. The time out limit for each SAT instance is 3600 seconds.

The run times of the different learning schemes on the benchmarks are shown in Table 3.5. The two tables are for VSIDS and the fixed decision heuristics respectively. Each benchmark class has multiple instances, shown in the parentheses following each benchmark name. In the results section, the times shown are the total time spent only on the instances that were successfully solved. The number of aborted instances is shown in the parentheses following run time. If there is no such number, all the instances are solved. For example, the barrel benchmark suite has 8 instances. Using the default VSIDS decision strategy, the All-UIP learning scheme took 1081.57 seconds to solve 7 of them, and aborted on one instance.

The experimental data show that for both default and fixed branching heuristics, the 1-UIP learning scheme clearly outperforms other learning schemes. Contrary to general belief, the decision schemes that generate small clauses, e.g. All-UIP and Decision do not perform as well.

Learning Scheme	Microprocessor Formal Verificaition			Bounded Model Checking			
	fvp-unsat.1.0(4)	sss.1.0(48)	sss.1.0a(9)	barrel(8)	longmult(16)	queueinvar(10)	satplan(20)
1-UIP	532.8	24.56	10.63	1012.62	2887.11	6.58	39.34
2-UIP	746.87	27.32	16.96	641.64	2734.57	16.37	41.37
3-UIP	2151.26	69.12	47.66	656.56	2946.73	19.44	57.16
alluip	0.68(3)	1746.27(2)	79.09	1081.57(1)	11160.25	18.07	71.86
relsat	2034.09	193.93	82.51	292.33(1)	5719.73	14.4	96.61
grasp	2224.44	94.64	33.99	654.54	6196.82	97.82	309.03
decision	0(4)	4490.18(15)	1356.44(2)	1685.73(1)	5315.26	3440.52(3)	2082.52(2)

(a) VSIDS Decision Scheme

1-UIP	11.36(3)	15307.13(3)	2997.37	281.48(1)	3141.8	817.07(5)	18(2)
2-UIP	23.07(3)	18844.51(3)	2646.99	344.34(1)	4279.07	777.2(5)	29.51(2)
3-UIP	40.75(3)	3985.23(9)	4109.86	432.77(1)	4440.49	860.3(5)	37.62(2)
alluip	0(4)	4063.44(25)	0.28(8)	699.42(1)	11375.32	2025.08(5)	1136.44(2)
relsat	80.94(3)	3114.83(16)	4261.25(4)	293.09(1)	4396.73	478.37(6)	3323.71(3)
grasp	22.51(3)	6497.99(8)	3382.47	326.46(1)	5597.1	792.12(5)	149.8(2)
decision	0(4)	2749.24(41)	40.57(8)	479.61(1)	7371.69	4933.99(6)	600.87(10)

(b) Fixed Decision Scheme

Table 3.5: Run time for different learning schemes (seconds)

Detailed statistics of the best of the proposed schemes (1-UIP) and the schemes employed in other SAT solvers (i.e. GRASP and relsat) are shown in Table 3.6. Three difficult test cases from three classes of the benchmark suites are shown in the table. The branching heuristic employed is VSIDS. The results in this table are quite interesting. As mentioned before, the GRASP scheme tries to learn as much as possible from conflicts. Actually, it will learn more than one clause for each conflict. For a given conflict, the clause learned by the 1-UIP scheme is also learned by the GRASP scheme. Therefore, in most cases, the GRASP scheme needs a smaller number of branches to solve the problem[5]. However, because the GRASP scheme records more clauses, the BCP process is slowed down. Therefore, the total run time of the GRASP scheme is more than that of the 1-UIP scheme. On the other hand, though relsat and 1-UIP both learn one clause for each conflict, the learned clause of relsat scheme is not as effective as 1-UIP, as it needs more branches.

From the experimental results it can be seen that different learning schemes can affect the behavior of the SAT solver significantly. The learning scheme based on the first Unique Implication Point of the implication graph (first UIP) has proved to be quite robust and effective in solving SAT problems in comparison with other schemes.

3.4 Cache Performances

In the previous sections, different BCP and learning schemes and their impact on SAT solvers have been examined. In this section, the perspective is changed a little bit. Another aspect of SAT solving, namely cache performance[6], is analyzed. It is well known that for current computer architectures, the memory hierarchy plays an important role in the performance of the computer systems. Algorithms designed

[5]For the bw_large case in the table, GRASP scheme needs more branches, probably because literals in the additional learned clauses adversely affect the VSIDS branching scheme.

[6]Part of the work described in this section was first reported in [142].

	1UIP	**relsat**	**GRASP**
Branches (10^6)	1.91	4.71	1.07
Added Clauses (10^6)	0.18	0.76	1.06
Added Literals (10^6)	73.13	292.08	622.96
Added lits/cls	405.08	384.31	587.41
Num. Implications (10^6)	69.85	266.21	78.49
Runtime (s)	522.26	1996.73	2173.14

(a) 9vliw_bp_mc (Unsatisfiable, 19148 variables)

Branches (10^6)	0.13	0.19	0.12
Added Clauses (10^6)	0.11	0.17	0.36
Added Literals (10^6)	17.16	28.07	77.47
Added lits/cls	150.17	162.96	213.43
Num. Implications (10^6)	71.72	108	74.83
Runtime (s)	339.99	612.64	762.94

(b) longmult10 (Unsatisfiable, 4852 variables)

Branches (10^6)	0.019	0.046	0.027
Added Clauses (10^6)	0.011	0.03	0.071
Added Literals (10^6)	0.56	1.61	6.12
Added lits/cls	50.87	52.94	86.19
Num. Implications (10^6)	7.23	17.84	16.81
Runtime (s)	23.44	52.64	124.13

(c) sat/bw_large.d (Satisfiable, 6325 variables)

Table 3.6: Detailed statistics of three large test cases for three learning schemes using VSIDS branching heuristic

with cache performance in mind can gain considerable speedup over non-cache aware implementations as demonstrated by many authors (e.g. [70, 133]). The importance of cache performance for SAT solvers was first discussed in [95]. Here, a cache analysis tool is used to quantitatively illustrate the gains achieved by a carefully designed algorithm and data structure.

The memory access pattern of a program depends on two things. The first is the algorithms involved, and the other is the implementation of the algorithms. Therefore, this section is divided into two main subsections. The first subsection examines the same solver employing different BCP mechanisms as described in Section 3.2. BCP takes the biggest portion of the run time for a typical SAT solver. This part shows how the BCP algorithms affect the cache behavior of the SAT solvers. In the second subsection, the experimental cache data on several different SAT solvers are shown. This is used to illustrate that careful data structure implementation can significantly improve the SAT solver's cache miss rate.

3.4.1 Cache Performances of Different BCP mechanisms

To evaluate the cache behavior of BCP mechanisms, the SAT solver zchaff [138] is modified. Zchaff already has data structures optimized for cache performance. Therefore, it can be argued that the data presented here is representative of the characteristics of each algorithm.

The instances used for the evaluation are chosen with various sizes and run times in order to avoid biasing the result. They include instances from microprocessor verification (2dlx), bounded model checking (bmc, barrel6), SAT planning (bw_large.d), combinational equivalence checking (c5315), DIMACS benchmarks (hanoi4), FPGA routing (too_large) as well as a random 3-SAT instance with clause to variable ratio of 4.2 (rand). Table 3.7 shows the statistics of the instances and the statistics of the solving process common to all of the deduction mechanisms. These statistics include

Instance Name	Num. Vars	Orig Cls	Orig Lits (k)	Lrned Cls	Lrned Lits (k)	Num. Impl (k)
2dlx_cc_mc_ex_bp_f	4583	41704	118	13756	1180	3041
barrel6	2306	8931	25	34416	2489	12385
bmc-galileo-9	63624	326999	833	2372	80	4629
bw_large.d	5886	122412	273	14899	399	11110
c5315	5399	15024	35	83002	7061	30174
hanoi4	718	4934	12	4578	259	364
rand	200	830	2	25326	500	1671
too_largefs3w8v262	2946	50416	271	71772	2847	9466

Table 3.7: Instances used to evaluate the BCP mechanisms

number of variables, the original and learned number of clauses and literals. The table also shows the number of implications needed to solve the problem. All the experiments were carried out on a Dell PowerEdge 1500sc computer with 1.13Ghz PIII CPU and 1G main memory. The PIII CPU has a separate level-1 cache comprised of a 16K data cache and a 16K instruction cache. The level-2 cache is a unified 512K cache. Both L1 and L2 caches have 32byte cache lines with 4-way set associativity.

The open source cache simulation and memory debugging tool `valgrind` [116] is used to perform the cache simulation. The simulated CPU is the same as the CPU the SAT solver runs on. The results are shown in Table 3.8 and Table 3.9. The tables show the real world run time (not simulation time) to solve each instance, number of instructions executed (in billions), number of data accesses (in billions), and cache miss rates for data accesses for each of the five BCP mechanisms described in Section 3.2. The cache misses for instructions are usually negligible for SAT solvers, and are therefore not shown in the table[7].

[7]Our implementation of the Head/Tail scheme may not be the best that can be achieved and may not be completely indicative of the actual performance of the scheme with optimized implementation.

Instance Name	**RunTime (seconds)**	10^9 **Instr. Executed**	10^9 **Data Accesses**	**L1 Data Miss Rate**	**L2 Data Miss Rate**
2dlx_cc_mc_ex_bp_f	20.09	10.24	4.78	13.54%	16.02%
barrel6	102.15	48.04	22.49	15.09%	16.61%
bmc-galileo-9	16.60	6.83	3.26	5.69%	56.87%
bw_large.d	118.71	28.63	12.82	14.44%	47.16%
c5315	427.47	169.53	81.21	16.48%	23.39%
hanoi4	1.79	1.63	0.78	12.50%	1.16%
rand_1	96.28	39.20	18.36	19.76%	19.09%
too_largefs3w8v262	730.49	171.81	79.58	20.34%	36.73%

(a) 2-Counter Scheme

2dlx_cc_mc_ex_bp_f	14.47	8.43	4.26	8.76%	15.20%
barrel6	67.61	37.78	19.52	9.08%	15.98%
bmc-galileo-9	12.25	6.43	3.17	3.49%	56.57%
bw_large.d	49.09	22.53	11.06	7.22%	28.87%
c5315	268.18	123.82	66.55	11.91%	21.28%
hanoi4	1.47	1.31	0.68	8.32%	1.59%
rand_1	46.73	23.70	13.20	13.22%	13.00%
too_largefs3w8v262	329.31	103.99	57.29	13.45%	32.25%

(b) 1-Counter Scheme

2dlx_cc_mc_ex_bp_f	12.74	8.12	4.31	5.86%	17.48%
barrel6	52.88	36.18	19.77	6.07%	13.69%
bmc-galileo-9	11.55	6.36	3.20	3.22%	55.21%
bw_large.d	39.84	21.97	11.21	5.23%	27.32%
c5315	172.82	116.22	66.92	7.13%	18.10%
hanoi4	1.35	1.25	0.69	4.13%	2.42%
rand_1	22.67	21.81	13.22	6.33%	7.95%
too_largefs3w8v262	141.19	96.29	57.48	8.29%	18.93%

(c) Compact 1-Counter Scheme

Table 3.8: Memory behavior of different BCP mechanisms: counter based

Instance Name	**RunTime (seconds)**	10^9 **Instr. Executed**	10^9 **Data Accesses**	**L1 Data Miss Rate**	**L2 Data Miss Rate**
2dlx_cc_mc_ex_bp_f	18.50	9.21	3.69	6.08%	28.98%
barrel6	82.38	42.12	16.65	5.83%	27.70%
bmc-galileo-9	16.17	7.52	3.51	3.69%	60.86%
bw_large.d	67.45	26.66	11.50	5.63%	41.15%
c5315	227.08	94.12	37.93	6.48%	35.53%
hanoi4	1.75	1.39	0.55	5.71%	7.09%
rand_1	40.95	20.11	8.64	7.10%	24.76%
too_largefs3w8v262	278.71	105.67	44.39	5.99%	51.69%

(d) Head/Tail Scheme

2dlx_cc_mc_ex_bp_f	7.26	5.48	2.13	3.83%	14.52%
barrel6	30.35	21.83	8.31	3.79%	12.90%
bmc-galileo-9	8.12	5.90	2.68	2.16%	54.37%
bw_large.d	25.36	17.10	6.97	3.44%	19.82%
c5315	66.70	45.20	17.78	4.27%	14.44%
hanoi4	0.87	0.79	0.31	3.25%	2.75%
rand_1	5.35	3.78	1.58	4.17%	7.66%
too_largefs3w8v262	41.62	25.99	10.28	3.85%	22.83%

(e) 2 Literal Watching Scheme

Table 3.9: Memory behavior of different BCP mechanisms: pointer based

The tables show that different implementations of the BCP mechanism have significant impact on the cache behavior of the SAT solver. Comparing the 2-Counter Scheme and the 1-Counter Scheme, the 1-Counter scheme reduces memory accesses by a small amount, but the cache miss rate for the 1-Counter scheme is significantly smaller than the 2-Counter scheme. This validates the intuition that the update of counters is the main cause of cache misses. By reducing the counter update by half, the cache miss rate is reduced by almost a half. Because the 1-Counter Scheme needs more checks for unit clauses compared with the 2-Counter, the total memory accesses are not reduced by as much. However, these memory accesses do not cause many misses.

Comparing the Compact 1-Counter Scheme with the regular 1-Counter Scheme, even though the number of instructions executed and the number of data accesses are almost the same, the locality of counter accesses improves the cache miss rate of the Compact 1-Counter Scheme significantly. The actual run time is also improved by as much as 2x. This again validates the previous conjecture, i.e. cache misses happen mostly on counter updates. As expected, the pointer based BCP schemes usually have a lower cache miss rate compared with the counter based schemes. The 2-Literal Watching scheme performs best among these schemes, with a significantly smaller number of instructions executed, as well as the lowest number of data accesses. Moreover, it has the lowest cache miss rate among these BCP mechanisms.

3.4.2 Cache Performances of Different SAT Solvers

In the previous subsection, several different BCP mechanisms are evaluated. Their respective cache performances are shown. In this section, the cache behavior of several existing SAT solvers is evaluated. These solvers are all based on the same DLL algorithm with learning and non-chronological backtracking [88, 64]. In the past, SAT solvers were often designed as a validation for various algorithms proposed by

Instance Name	Num. Variables	Num. Clauses	Num. Literals
x1_40	118	314	940
rand_2	400	1680	5028
longmult12	5974	18645	44907
barrel9	8903	36606	102370
9vliw_bp_mc	20093	179492	507734
5pipe	9471	195452	569976

Table 3.10: Instances for cache miss experiments for SAT solvers

academic researchers. Therefore, not much emphasis was put on implementation issues. Recently, as SAT solvers have become more widely used as a viable deduction engine, there is great emphasis on implementing them efficiently.

The solvers being evaluated include some of the most widely used SAT solvers such as GRASP [88], relsat [64] and SATO [136]. The SAT solver zchaff [138] is also included in the evaluation. Due to different branching heuristics used, these solvers behave drastically differently on different benchmark instances. Therefore, it is not fair to compare their running time for solving a small number of SAT instances. In order to concentrate on the cache behavior of the solvers, some difficult SAT instances are chosen and each solver run under `valgrind` for 1000 seconds[8]. None of the solvers can solve any of the test instances during the run time. This enable us to focus on cache behavior independent of the total run time. Except for relsat, all other solvers are run with default parameters. The option `-p0` is enabled on relsat to disable the time consuming preprocessing. The benchmarks used include SAT instances generated from bounded model checking [11] (barrel, longmult), microprocessor verification [132] (9vliw, 5pipe), random 3-SAT (rand2) and equivalence checking of xor chains (x1_40). These benchmarks used for evaluating cache miss rates for the solvers are listed in Table 3.10.

[8]Programs running under valgrind are around 30 times slower than on the native machine.

Instance Name	GRASP		SATO		relsat		zchaff	
	L1%	**L2%**	**L1%**	**L2%**	**L1%**	**L2%**	**L1%**	**L2%**
x1_40	23.72	75.22	1.72	0.00	1.73	0.01	3.32	26.32
rand_2	25.05	60.38	36.79	0.57	11.14	0.12	6.58	23.46
longmult12	20.86	57.69	19.32	41.68	13.24	19.78	6.46	18.17
barrel9	21.22	51.73	21.62	55.55	15.73	19.14	6.62	33.43
9vliw_bp_mc	22.43	88.58	40.49	42.85	14.58	55.34	9.48	54.37
5pipe	31.14	81.74	36.22	4.22	15.38	33.39	6.14	21.54

Table 3.11: Data cache miss rates for different SAT solvers

The results of cache miss rates for different SAT solvers are listed in Table 3.11. The table shows that compared with zchaff, SAT solvers such as GRASP, relsat and SATO perform poorly in memory behavior. One of the reasons for the high cache miss rate of GRASP and relsat is due to their use of a counter-based BCP mechanism. But comparing their cache miss rates with the best that can be achieved by counter-based BCP mechanisms as shown in previous section, it can be observed that they are still under par, even considering their BCP mechanism[9]. SATO use the same BCP mechanism as the Head/Tail mechanism, but it has a much higher miss rate than a similar BCP scheme implemented on zchaff. One benchmark for which SATO performs well is the xor chain verification instance x1_40. The reason for this is that x1_40 is a very small instance and SATO has a very aggressive clause deletion heuristic. Therefore, SATO can fit the clauses into the L1 cache while other solvers cannot do so due to their more tolerant clause deletion policy. The same reason explains the low cache miss rate for relsat on x1_40, and both SATO's and relsat's low L2 miss rate for the instance rand_2.

The reasons for zchaff achieving a much lower cache miss rate than other SAT solvers are the following implementation and algorithmic improvements:

[9]The benchmarks used are not exactly the same for these two experiments, but still the numbers are representative of their respective behaviors.

1. **Better BCP Algorithms**

   The *2-Literal Watching* BCP scheme [95] used in zchaff has a much better cache miss rate than counter based schemes used by GRASP and relsat, as proven by the experimental results of Tables 3.8 and 3.8.

2. **Pointer Free Data Structure**

   Zchaff avoids the use of pointers in the data structure for the clause database. Unlike other SAT solvers, the linked lists are avoided in order to increase memory locality. Instead, they are replaced by arrays. Arrays are less flexible than linked lists because clause deletion and insertion take more effort to implement. But the memory access for array data structures are more localized.

3. **Careful Implementation**

   The SAT solver zchaff is tuned by extensive profiling. Much effort has been spent on trying to code it as efficiently as possible. Some of the existing SAT solvers are simply not well implemented and therefore less efficient.

3.5 Chapter Summary

This chapter describes the work on the analysis and efficient implementation of SAT algorithms based on the Davis Logemann Loveland (DLL) search algorithm. Traditionally, SAT solver research has mainly focused on algorithmic improvements. In the past, tremendous progress has been made on reducing the search tree size with smart branching heuristics, intelligent learning schemes and powerful reasoning methods. This chapter examines the problem from a different perspective. The results in this chapter show that the implementation aspects of SAT solvers are equally important. Wise choice of heuristics, careful tuning of parameters and efficient coding can greatly improve the overall performance of a SAT solver as well.

Chapter 4

The Correctness of SAT Solvers and Related Issues

Because of recent improvements in SAT solving techniques, the efficiency and capacity of state-of-the-art SAT solvers have improved dramatically. Consequently SAT solvers are being increasingly used in industrial applications as the kernel deduction mechanism [12, 27]. In real world applications, speed is only part of the requirements for SAT solvers. In addition, some special requirements may be needed for SAT solvers be be acceptable for certain applications. This chapter discuss some capabilities of SAT solvers that can facilitate the deployment of SAT solvers for some real world applications.

Automatic reasoning tools have been used for mission critical applications such as railway station safety checking [47]. In these kinds of applications, the users often require that the reasoning tools provide a checkable proof during the reasoning process so that results can be independently verified by a third party checker. As SAT solvers become more popular, such a feature is also required for modern SAT solvers. This is discussed in Section 4.4.

Given an unsatisfiable Boolean formula in Conjunctive Normal Form, sometimes it may be desirable to determine a subset of the original clauses such that the formula

obtained by the conjunction of the clauses in the subset is still unsatisfiable. This task is called "extracting the unsatisfiable core of an unsatisfiable formula". The motivation as well as an implementation for this is discussed in detail in Section 4.5.

The two tasks mentioned above may seem to be unrelated at first glance. However, they are actually linked with each other as will be shown in this chapter. In Section 4.2, another seemingly unrelated issue of the SAT solving algorithm is discussed. The section provides a formal proof that the DLL algorithm with non-chronological backtracking and learning employed by modern SAT solvers is sound and complete. Then Section 4.3 shows why this proof is interesting and can be used as a foundation for the two tasks mentioned above. The section introduces the notion of a *resolution graph* for a proof of unsatisfiability and illustrates how to accomplish the above mentioned tasks by traversing this graph.

In order to facilitate the discussion, The DLL algorithm with learning and non-chronological backtracking is re-discussed in detail in Section 4.1. This algorithm (or minor variations of it) is employed in almost all modern SAT solvers. This section illustrates that the learning process discussed in the previous two chapters can be formulated as a resolution process. In Section 4.6, some related contemporary work is discussed. Finally, the chapter is summarized in Section 4.7.

4.1 The Algorithm of a DLL SAT Solver

This section briefly reviews the algorithm employed by a typical SAT solver. This algorithm is exactly the same algorithm used by the SAT solver chaff [95]. It (or a minor modification of it) is the algorithm employed by all modern SAT solvers such as GRASP [88], relsat [64], and BerkMin [44]. The top level pseudo-code of the algorithm is listed in Figure 4.1. This is essentially the same DLL algorithm originally shown in Figure 2.5, with some minor modifications to facilitate the analysis.

```
DLL()
{
    if (preprocess()==CONFLICT)
        return UNSATISFIABLE;
    while(1) {
        if (decide_next_branch()) {      //Branching
            while(deduce()==CONFLICT) {    //Deducing
                blevel = analyze_conflicts();//Learning
                if (blevel < 0)
                    return UNSATISFIABLE;
                else back_track(blevel); //Backtracking
            }
        }
        else                 //no free variables left
            return SATISFIABLE;
    }
}
```

Figure 4.1: The top-level algorithm of DLL with learning and non-chronological backtracking

Here the control flow of the algorithm is discussed in more detail. If the preprocessor cannot determine the result for this instance, the solver will go into the main loop. Function `decide_next_branch()` selects a free variable to branch. If no free variable exists, then the instance must be satisfiable because all variables have been assigned a value and no conflicting clause exists. Therefore, the solver will return `SATISFIABLE`. Otherwise, deduction will be performed after the branch. If conflict exists, the conflict analysis engine will be called. If the solver determines that the backtracking level is smaller than 0, it means that there exists conflict even without any branch. In that case, the solver will report the instance to be unsatisfiable and exit.

The method of conflict analysis and the way to determine the backtracking level is crucial to the completeness of the solver, so it will be discussed here in a little more

```
analyze_conflict() {
    if (current_decision_level()==0)
        return -1;
    cl = find_conflicting_clause();
    do {
        lit = choose_literal(cl);
        var = variable_of_literal(lit);
        ante = antecedent(var);
        cl = resolve(cl, ante, var);
    } while (!stop_criterion_met(cl));
    add_clause_to_database(cl);
    back_dl = clause_asserting_level(cl);
    return back_dl;
}
```

Figure 4.2: Conflict analysis by iterative resolution

detail. Here the conflict analysis procedure is formulated as a resolution process. Resolution has been defined in Chapter 2. Two clauses can be resolved as long as the distance between the two clauses is one.

The pseudo-code for function `analyze_conflict()` is listed in Figure 4.2. At the beginning, the function checks whether the current decision level is already 0. If that is the case, the function will return -1, essentially saying that there is no way to resolve the conflict and the formula is unsatisfiable. Otherwise, the solver determines the conflicting clause. Iteratively, the procedure resolves the conflicting clause with the antecedent clause of a variable that appears in it. Function `choose_literal()` always chooses a literal in reverse chronological order, meaning that it will choose the literal in the clause that is assigned last. Function `resolve(cl1,cl2,var)` will return a clause that has all the literals appearing in `cl1` and `cl2` except for the literals corresponding to `var`. Notice that the conflicting clause has all literals evaluating to *false*, and the antecedent clause has all but one literal evaluating to *false* (because it

is a unit clause) and the remaining one literal evaluates to *true*. Therefore, one and only one variable appears in both clauses with different phases and the two clauses can be resolved. The resolvent clause is still a conflicting clause because all its literals evaluate to *false*, and the resolution process can continue iteratively.

The iterative resolution will stop if the resulting clause meets a stopping criterion. The stopping criterion is that the resulting clause be an *asserting clause*. An asserting clause is a clause with all false literals, and among them only one literal is at the current decision level and all the others are assigned at a decision level less than current. In Chapter 3, the technique of non-chronological backtracking and conflict-driven learning is described by analyzing the implication graph and considering cuts that bipartition this graph. In Figure 4.3, an example is used to illustrate why a cut in the implication graph and an iterative resolution process are equivalent. The learned clause is generated by the 1-UIP cut shown in the figure. Clauses 1 and 2 are the original clauses in the clause database that correspond to the antecedent clauses of variables V_4 and V_1 respectively. Clause 3 is the conflicting clause. By following the iterative resolution process described in Figure 4.2, first the Clause 4 is generated, which is the resolvent of Clauses 3 and 2. By continuing the resolution, Clause 5 is obtained, which is the resolvent of Clauses 4 and 1. Since Clause 5 satisfies the stopping criterion of being an asserting clause, the loop will terminate. It can be seen that Clause 5, which is generated by the iterative resolution process, is exactly the same as the clause corresponding to the 1-UIP cut in the implication graph.

After backtracking, the asserting clause will be a unit clause and the unit literal will be forced to assume the opposite value, thus bringing the search to a new space. The flipped variable will assume the highest decision level of the rest of the literals in the asserting clause. This decision level is called the *asserting level*. After the asserting level is found, function `backtrack()` will undo all the variable assignments between the current decision level and the asserting level, and reduce the current decision level

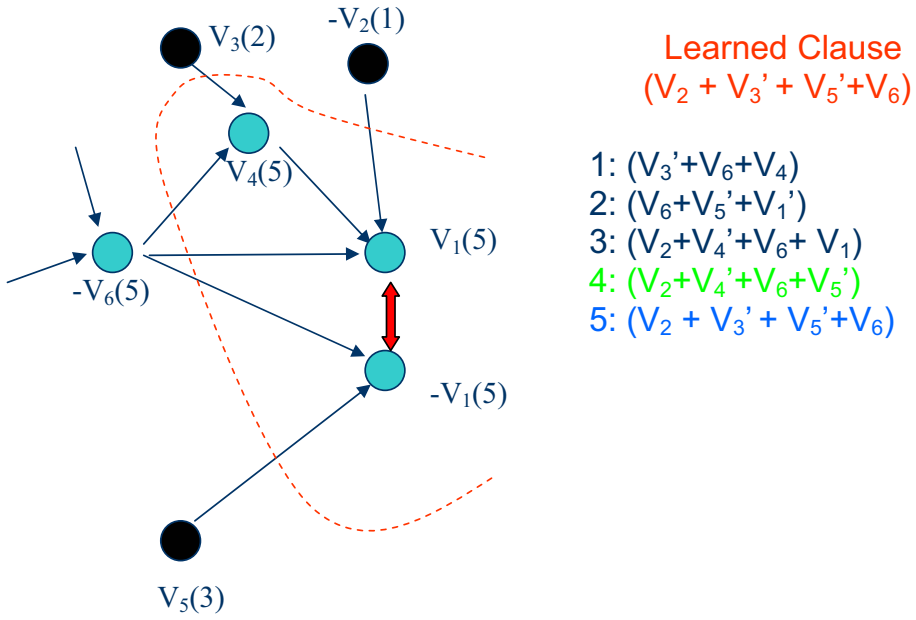


Figure 4.3: Learning as a resolution process

to asserting level accordingly. Such a backtracking scheme will be called *assertion-based backtracking*. In assertion-based backtracking, no explicit "flip" operation for decision variables are needed, and the solver does not need to remember whether a decision variable has been visited in both phases or not. As shown in the following sections, this backtracking scheme is the key to the results in the rest of this chapter.

The resulting clauses from the resolution can be optionally added back to the clause database because they are consistent with the existing clauses and adding them would not change the satisfiability of the SAT formula. This mechanism is called *learning*. Learning is not required for the correctness or completeness of the algorithm. Experiments show that learning can often help prune future search space and reduce solving time, therefore it is always employed in modern SAT solvers. Learned clauses can also be deleted in the future if necessary. Regardless of whether the solver employs learning or not, the clauses that are antecedents of currently assigned variables should always be kept by the solver because they may be used in

the future resolution process.

4.2 The Correctness of the DLL Algorithm

This section will prove that the algorithm described in Section 4.1 is indeed sound and complete. This forms the basis for the checker and the unsatisfiable core extractor discussed in the following sections. A SAT algorithm is correct if given enough run time the solver can always determine the correct satisfiability of the input formula. The correctness of the algorithm is proved using the following proposition due to Davis and Putnam [32].

Proposition 2: *A propositional Boolean formula in CNF is unsatisfiable if it is possible to generate an empty clause by resolution from the original clauses (see, e.g. [32]).*

Proof: Clauses generated from resolution are redundant and can be added back to the original clause database. If a CNF formula has an empty clause in it, then it is unsatisfiable. Therefore, the proposition holds. ■

The correctness of the algorithm described in Figure 4.1 is due to the following three lemmas.

Lemma 1: *Given enough run time, the algorithm described in Section 4.1 will always terminate.*

Proof: Given a variable assignment state during the search process, let $k(l)$ denote the number of variables assigned at decision level l. Suppose the input formula contains n variables. Because at least one variable is assigned at each decision level except decision level 0, It is obvious that

$$\forall l, 0 \leq l \leq n, k(l) \leq n;$$

$$\forall l > n, k(l) = 0;$$

$$\sum_{l=0}^{n} k(l) = n;$$

Consider a function f:

$$f = \sum_{l=0}^{n} \frac{k(l)}{(n+1)^l}$$

The function is constructed such that for two different variable assignment states α and β of the same propositional formula, the value $f_\alpha > f_\beta$ if and only if there exists a decision level $d, 0 \leq d < n$,

$$k_\alpha(d) > k_\beta(d);$$

and for any l such that $0 \leq l < d$,

$$k_\alpha(l) = k_\beta(l)$$

Intuitively, this function biases the weight towards the number of variables assigned at lower decision levels. Note that the following inequality holds:

$$\frac{1}{(n+1)^j} > \frac{n}{(n+1)^{j+1}}$$

Therefore, for two different variable assignment states, the one with assignments more "concentrated" at the lower decision levels has a larger value of f.

Now consider function f during the search process. Its value monotonically increases as the search progresses. This is obvious when no conflict exists, because the solver keeps assigning new variables at the highest decision level without changing the previous assignments. When a conflict occurs, assertion-based backtracking moves an assigned variable at the current decision level to the asserting level, which is smaller than the current decision level. Given the heavy bias of f, regardless of the subsequent un-assignments of variables between asserting level and current decision level, the value of f still increases compared with its value before the conflict analysis.

Because f can take only a finite number of different values, the algorithm is guaranteed to terminate after sufficient run time. ■

Notice that the proof for termination does not require the solver to keep all learned clauses. The value of function f will monotonically increase regardless of whether any learned clauses are added or deleted. Therefore, contrary to common belief, deleting learned clauses cannot cause the solving process to loop indefinitely. On the other hand, the often employed mechanism called restart [46] actually invalidate the proof. The restart mechanism undoes all the assignments of a search tree once in a while and restarts the search from the beginning. After each restart, all the variables are unassigned, the value of f decreases. In fact, if learned clauses are deleted and restart is enabled during the solving process, it is possible to make the solver loop indefinitely. Therefore, it is important for the solver to increase the restart period as the solving progresses to make sure that the algorithm terminates.

Lemma 2: *If the algorithm described in Section 4.1 returns* `SATISFIABLE`*, then the formula is satisfiable, i.e. there exists a variable assignment that makes the formula evaluate to* true*.*

Proof: The algorithm in Figure 4.1 will return `SATISFIABLE` if and only if the function `decide_next_branch()` determines that all variables are assigned a value and no free variable is available for branching. If all variables are assigned a value and no conflict exists (because none is found by function `deduce()`), the formula is obviously satisfied. ■

Lemma 3: *If the algorithm described in Section 4.1 returns* `UNSATISFIABLE`*, then the formula is unsatisfiable.*

Proof: This lemma is proved using Proposition 2. The proof will show how to construct an empty clause from the original clauses by resolution.

When the solver returns `UNSATISFIABLE`, the last `conflict_anaysis()` function must have encountered a conflicting clause while the current decision level is 0. Starting from that clause, it is possible to iteratively perform resolutions similar to the `while` loop shown in Figure 4.2. This loop can continue as long as there are literals in the resulting clause. Because there is no decision variable at decision level 0, every assigned variable must have an antecedent. Since function `choose_literal()` chooses literals in reverse chronological order, no literal can be chosen twice in the iteration and the process will stop after at most n calls to `resolve()`. At that point, an empty clause is generated. All the clauses involved in the resolution process are either original clauses or learned clauses that are generated by resolution from the original clauses; therefore by Proposition 2, the propositional formula is indeed unsatisfiable. ■

Lemma 1 proves that the algorithm will terminate given enough run time. Lemmas 2 and 3 prove that the algorithm will return the correct result for the satisfiability of the formula. Therefore, this algorithm is sound and complete. Therefore, the following theorem follows directly.

Theorem 1: *The algorithm depicted in Figure 4.1 is sound and complete.*

4.3 Resolution Graph

The proof of Lemma 3 shows that given an unsatisfiable Boolean formula, a SAT solver based on DLL can generate an empty clause from the original clauses using resolution. To formalize this idea, this section introduces the notion of a *resolution graph* for an unsatisfiable proof. An example of a resolution graph is shown in Figure 4.4. A resolution graph is a directed acyclic graph (DAG) that represents a SAT solving process of a DLL SAT solver for a Boolean formula. Each node in the graph represents a

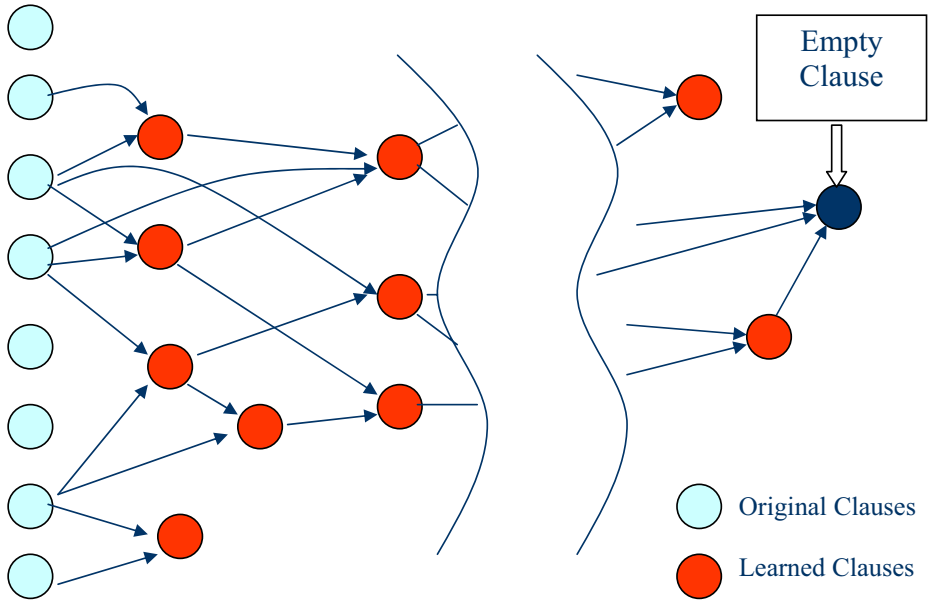


Figure 4.4: The resolution graph

clause. The source nodes (the nodes that do not have incoming edges) are the original clauses in the Boolean formula. The internal nodes represent the learned clauses generated during the solving process. The edges in the resolution graph represent resolution. If an edge from node b to node a appears in the resolution graph, the node b (or the clause represented by b) is said to be a *resolve source* of node a (or the clause represented by a). A clause represented by an internal node is obtained in the solving process by resolving all clauses that are resolve sources of that clause one by one. In the conflict analysis procedure depicted in Figure 4.2, the resolve sources of the final learned clause (the clause added to the database by `add_clause_to_database(cl)`) are all the clauses that corresponds to `ante` in the pseudo-code together with the conflicting clause obtained by `find_conflicting_clause()`.

As mentioned in the last subsection, if the formula is proven to be unsatisfiable, there exists a resolution process to generate an empty clause. The empty clause is shown in Figure 4.4. The resolve sources of the empty clause are the final conflicting

clause together with the antecedent clauses of variables assigned at decision level 0. It is easy to see that a resolution graph that represents a proof of the Boolean formula being unsatisfiable contains an empty clause in it, and vice versa.

It should be pointed out that even though the resolution graph discussed in this chapter are generated from a SAT solver that is based on DLL with an assertion-based backtracking mechanism, the concept of a resolution graph itself is not limited to such SAT solvers. As shown by [130], a wide variety of SAT solving techniques, such as look-ahead decisions [75] and regular DLL without learning, can also be viewed as a resolution process. Proofs generated by SAT solvers employing these techniques also have corresponding resolution graphs. As long as the resolution graphs can be extracted, the techniques discussed in the subsequent sections will be applicable. On the other hand, for some other reasoning techniques, such as BDDs [15] and equivalence reasoning [74], it may not be possible to formulate them as a resolution process[1]. In the rest of the chapter, only SAT solvers that are based on the DLL algorithm with assertion-based backtracking are considered since it is the most widely employed algorithm in modern SAT solvers.

4.4 Checking the Unsatisfiable Proof

Typically[2], a SAT solver takes a propositional formula as input and produces an output that claims the formula to be either satisfiable or unsatisfiable. A formula is satisfiable if there exists a variable assignment that makes the formula evaluate to *true*; a formula is unsatisfiable if no such assignment exists. When the solver claims that an instance is satisfiable, it is usually possible for the solver to provide a

[1]It is not to say that the formula cannot be proven to be unsatisfiable by resolution. Instead, the resolution graph is not a by-product of the proof for solvers using these techniques. For example, when a formula is proven to be unsatisfiable using BDD technique, it is hard to extract a resolution graph from the proof.

[2]Part of the work described in this section was first reported in DATE 2003[144].

satisfying variable assignment with very little overhead. A third party program can easily take the assignment and verify if it indeed makes the formula evaluate to *true*. Because of the NP-Complete nature of SAT, it is guaranteed that such a check takes polynomial time with regard to the size of the SAT instance. In fact, in practice the Boolean formulae provided to the solver are usually presented in Conjunctive Normal Form (CNF). In that case, this check takes linear time to finish, essentially making such a check trivial to implement.

On the other hand, when a SAT solver reports that a Boolean formula is unsatisfiable, it is usually not trivial for a third party to verify the correctness of that claim. In Section 4.2, the DLL algorithm employed by modern SAT solvers is proven to be indeed sound and complete. This is hardly a surprise. However, a SAT solver is a fairly complex piece of code, and often bugs may appear in the code and affect the correctness of the solver. In fact, during the recent SAT 2002 solver competition, several submitted SAT solvers were found to be buggy [120].

Previously, in practice, people often ran multiple solvers on the same formula and compared the results. This is often not a viable solution in practice because several SAT instances are too hard to solve by all but the most powerful SAT solver. Moreover, such checks will take a significant amount of time, which is comparable to the original SAT solving process. It is also possible (though unlikely) that though multiple solvers agree on the results, they may all be wrong. Therefore, a more rigorous and formal way is needed to check that the SAT solver's claim of unsatisfiability of the formula is indeed true.

The idea of requiring automated reasoning tools to produce easily checkable proofs is not new. Some theorem provers (e.g. [127]) and model checkers (e.g. [98]) have this ability. Ideas about verifying the correctness of SAT solvers have been proposed by other authors also. In a paper about Stålmarck's algorithm [54], the author mentions that one of the requirements for Stålmarck's SAT solver is being able to provide a

proof trace that can be independently checked. However, the author does not provide the details about the checking procedure. The SAT solver in discussion is based on proprietary algorithms, so the checking procedure may not be applicable to the current crop of efficient SAT solvers which are based on the DLL algorithm [31]. Van Gelder [130] showed how a DLL based SAT procedure can produce easily checkable resolution-based proofs. However, in that work the solver discussed is based on the concept of "flipping" decision variables. This is not the case for most of the modern SAT solvers, which are based on "assertion". Therefore, its unclear how can the method be adapted to the current crop of SAT solvers such as chaff [95] and Berk-Min [44]. Moreover, in that work a checker is not provided, thus no evaluation of the actual feasibility of the checking process is provided.

This section describes a checker that is derived from the results of the previous section. The checker can prove a formula to be unsatisfiable when a SAT solver claims so. The checker can also provide a lot of information for debugging purposes when a solver produces erroneous results. In fact, one of the motivations for this work was an intricate bug in an earlier version of zchaff that was very hard to locate. The solver claimed a satisfiable instance to be unsatisfiable after hours of computation. With the help of the checker, the bug was quickly located and fixed.

This section is divided into three subsections. Section 4.4.1 describe the modifications that are needed for the SAT solver to produce a trace file for the checker. Section 4.4.2 and Section 4.4.3 describe two different approaches for the implementation of the checker.

4.4.1 Producing the Trace for the Checker

For the checker to be able to verify that a SAT instance is indeed unsatisfiable, it only needs to verify that there exists a sequence of resolutions among the original clauses that can generate an empty clause. To achieve this, the SAT solver can be modified

to produce some information as a trace to aid the resolution checking process of the checker. As mentioned before, an unsatisfiable proof from a DLL SAT solver has a corresponding resolution graph that contains an empty clause. Therefore, the SAT solver can produce a trace file that contains enough information for the checker to reconstruct the resolution graph and check if the empty clause can really be generated by resolution from the original clauses.

In this approach, each clause encountered in the solving process is assigned with a unique ID. There is no requirement on how the ID should be assigned, as long as both the solver and the checker agree to use the same ID for the same clause in the original CNF formula. In the actual implementation, the ID of an original clause is equal to the number of clauses appearing before it in the input file that contains the propositional CNF formula. These IDs are used to identify clauses in the trace produced by the solver.

The trace is produced by modifying the solver in the following ways:

1. Each time a learned clause is generated, the learned clause ID is recorded, together with the clauses that are involved in generating this clause. In Figure 4.2, these clauses include the clause returned by `find_conflicting_clause()`, as well as all the clauses that corresponds to parameter `ante` in the calls to `resolve()`. These clauses are the *resolve sources* of the generated clause.

2. When `analyze_conflict()` is called and current decision level is 0, the solver will record the IDs of the clauses that are conflicting at that time before returning value -1. These clauses are the *final conflicting clauses*. Different final conflicting clauses correspond to different proofs of the unsatisfiability of the formula. In the proof process, only one of these final conflicting clauses are needed to construct a proof, so it is sufficient to record only one ID.

3. When the solver determines that `analyze_conflict()` returns with value -1, before returning with value `UNSATISFIABLE`, the solver will record all the variables that are assigned at decision level 0 together with their values and the IDs of their antecedent clauses in the trace file.

These modifications total less than twenty lines of C++ code in the SAT solver zchaff, and should be very easy to make for other SAT solvers based on the same assertion-based backtracking technique such as GRASP [88] and BerkMin [44].

The checker will take the trace produced together with the original CNF formula and try to generate an empty clause in a manner similar to the description of the proof for Lemma 3 in Section 4.2. However, initially the checker does not have the actual literals of the learned clauses. Instead, from the trace file the checker only knows the resolve sources that are used to generate each of these learned clauses[3]. Therefore, to accomplish the resolution process, the checker needs to use resolution to produce the literals in the needed learned clauses first. There are two different approaches for the process of producing the needed clauses. Section 4.4.2 describes a depth-first approach for the traversal. Section 4.4.3 describes a breadth-first approach for the traversal.

4.4.2 The Depth-First Approach

The depth-first approach for building the learned clauses traverse the resolution graph in a depth-first way. It begins from the final conflicting clause and builds the needed clauses by resolution recursively. It is essentially a lazy scheme in the sense that it will only try to build a clause if the clause is needed in the resolution process to produce the final empty clause. The algorithm for this approach is described in Figure 4.5.

[3]Notice that it is useless to record the literals in the learned clauses into the trace because the checker cannot assume the validity of the learned clauses in the first place. To guarantee the validity of the learned clauses, the checker essentially needs to go through the same process of checking the proof.

```
check_depth_first()
{
    id = get_one_final_conf_clause_id();
    cl = recursive_build(id);
    while(!is_empty_clause(cl)){
        lit = choose_literal(cl);
        var = variable_of_literal(lit);
        ante_id = antecedent_id(var);
        ante_cl = recursive_build(ante_id);
        cl = resolve(cl,ante_cl);
    }
    if (error_exists())
        printf("Check Failed");
    else
        printf("Check Succeeded");
}

recursive_build(cl_id)
{
    if (is_built(cl_id))
        return actual_clause(cl_id);
    id = get_first_resolve_source(cl_id);
    cl = recursive_build(id);
    id = get_next_resolve_source(cl_id);
    do {
        cl1 = recursive_build(id);
        cl = resolve(cl,cl1);
        id = get_next_resolve_source(cl_id);
    }while(id != NULL);
    associate_clause_with_id(cl, cl_id);
}
```

Figure 4.5: The depth-first approach of the checker

Function `check_depth_first()` implements the algorithm described in the proof for Lemma 3 in Section 4.2. It begins with one final conflicting clause (which is recorded in the trace) and tries to generate an empty clause by resolution. If a clause is needed, it will be built on the fly. This operation is accomplished by the function `recursive_build()`. The function takes a clause ID as an input, and checks if it has the literals corresponding to that clause ID or not. If not, it will build the clause and associate the clause ID with the resulting literals.

Because the algorithm described in Section 4.1 has already been proven to be correct, therefore in the checking process the only possible result should be "Check Succeeded" if no bug exists in the implementation of the SAT solver. The actual checking is built in at each of the functions. For example, when `resolve(cl1, cl2)` is called, the function should check whether there is one and only one variable appearing in both clauses with different phases; when a variable's antecedent clause is needed, the function should check whether the clause is really the antecedent of the variable (i.e. whether it is a unit clause and whether the unit literal corresponds to the variable). If such checks fail, the functions will print out an error message and raise an error flag. The checker can try to provide as much information as possible about the failure to help debug the SAT solver.

The advantage of the depth-first approach is that the checker will only build the clauses that are involved in the empty clause generation process. If a learned clause is never used in that process, it will not be built by the checker. Because of this, the checking process is faster compared with the breadth-first approach that will be discussed in the next subsection.

The disadvantage of the depth-first approach is that in order to make the recursive function efficient, the checker needs to read in the entire trace file into main memory. As shown in the experimental section, the trace files can actually be quite large for some SAT instances and will not fit in memory. Therefore, for some long proofs, the

checker may not be able to complete the checking.

4.4.3 The Breadth-First Approach

To avoid the problem of memory blow up in a depth-first approach, the learned clauses can also be built with a breadth-first approach. The resolution graph is visited in a breadth-first manner. The clauses are built bottom-up so that when the checker builds a learned clause, all its resolve sources would have been built already. The checker traverses the learned clauses in the order in which they are generated during the solving process. This is the same order as the order of the clauses appearing in the trace file. The checker reads in the original clauses first. Then it will parse the trace file. Whenever a learned clause's resolve sources are read in from the trace file, its literals will be built by resolution. All the clauses that appear in its resolve sources should already be in the database because of the nature of the approach. In the end, when the checker constructs the empty clause, all clauses should be in the memory for resolution.

However, if the checker is implemented as described in the previous paragraph, it will still have the memory blow-up problem because the checker will have all the learned clauses in memory before the final empty clause construction begins. It is well known that storing all the learned clauses in memory is often not feasible (that is the motivation for learned clause deletion). Therefore, the checker needs to delete some clauses when it knows that they will not be needed anymore in the future resolution process. Note that because of the breadth-first nature, a clause can be deleted once its use as a resolve source is complete. Therefore, to solve the memory overflow problem, a first pass through the trace can determine the number of times a clause is used as a resolve source. During the resolution process, the checker tracks the number of times the clause has been used thus far as a resolve source and when its use is complete, the clause can be deleted safely from main memory.

In the actual implementation, the clause's total usage as a resolution source is stored in a temporary file because there is a possibility that even keeping just one counter for each learned clause in main memory is still not feasible. For the same reason, the first pass may also need to be broken into several passes so that the number of clauses as resolve sources can be counted in one range at a time. The checker is guaranteed to be able to check any proof produced by a SAT solver without danger of running out of memory because during the resolution process, the checker will never keep more clauses in memory than the SAT solver did when producing the trace. Because the SAT solver did not run out of memory (it finished and claimed the problem to be unsatisfiable), the checker will not run out of memory during the checking process either, assuming both programs run with same amount of memory.

4.4.4 Experiment Results

This section reports some experimental results of checking the proofs of unsatisfiability. The experiments are carried out on a PIII 1.13G machine with 1G memory. The memory limit is set to 800MB for the checkers. The benchmarks used are some relatively large unsatisfiable instances that are commonly used for benchmarking SAT solvers. They include instances generated from EDA applications such as microprocessor verification [132] (9vliw, 2dlx, 5pipe, 6pipe, 7pipe), bounded model checking [11] (longmult, barrel), FPGA routing [97] (too_largefs3w8v262) combinational equivalence checking (c7225, c5135) as well as a benchmark from AI planning [65] (bw_large). The statistics of the instances used are listed in Table 4.1. The run times to solve these instances using zchaff are also shown.

The SAT solver zchaff is modified to produce the trace file to represent the resolution graph that is needed for the checker. Table 4.2 shows the statistics of the resolution graph produced and run time overhead for producing the trace. Table 4.2 shows that the overhead for producing the trace file (1.7% to 12%) is not significant.

Instance Name	**Num. Variable**	**Num. Clauses**	**Proof Time (s)**
2dlx_cc_mc_ex_bp_f	4583	41704	3.3
bw_large.d	5886	122412	5.9
c5315	5399	15024	22.0
too_largefs3w8v262	2946	50216	40.6
c7552	7652	20423	64.4
5pipe_5_ooo	10113	240892	118.8
barrel9	8903	36606	238.2
longmult12	5974	18645	296.7
9vliw_bp_mc	20093	179492	376.0
6pipe_6_ooo	17064	545612	1252.4
6pipe	15800	394739	4106.7
7pipe	23910	751118	13672.8

Table 4.1: The unsatisfiable instances used for experiments

Also, it can be noticed that the overhead percentage tends to be small for difficult instances. The table shows the number of nodes and edges in the resolution graph that corresponds to the proof (node count excluding the original clauses and the empty clause, edge count excluding the incoming edges to the empty clause). It also shows the file sizes for the traces. The resolution graphs are big for the proofs and the trace files produced by the SAT solver are quite large for difficult benchmarks. For example, the instance 7pipe.cnf takes 13600 seconds to finish, and the trace file produced is 736 MB in size. It should be pointed out that the format of the trace file used is not very space-efficient in order to make the trace human readable. It is quite easy to modify the format to emphasize space efficiency and get a 2-3x compaction (e.g. use binary encoding instead of ASCII). By doing so, the efficiency of the checker is also expected to improve as profiling shows that a significant amount of run time for the checker is spent on parsing and translating the trace files.

Table 4.3 shows the statistics for the checkers to check the validity of the proofs.

Instance Name	**Trace Overhead**	**Trace Size (KB)**	**Resolution Graph**	
			#Nodes (k)	**#Edges (k)**
2dlx_cc_mc_ex_bp_f	11.89%	1261	9.6	184.1
bw_large.d	9.12%	1367	7.1	182.1
c5315	10.45%	11337	50.3	1951.8
too_largefs3w8v262	7.68%	8866	91.7	1227.8
c7552	8.76%	24327	100.5	4068.2
5pipe_5_ooo	4.51%	17466	79.8	2434.9
barrel9	4.51%	19656	121.1	3058.4
longmult12	6.17%	102397	131.6	18341.0
9vliw_bp_mc	4.26%	39538	255.6	5497.0
6pipe_6_ooo	3.39%	151858	462.1	21497.5
6pipe	2.77%	493655	1327.4	69102.9
7pipe	1.68%	736053	2613.9	93903.5

Table 4.2: Trace generation and statistics of the resolution graph

For the depth-first approach, the column "Cls Blt" shows the number of clauses that have their literals constructed by function `recursive_build()` in Figure 4.5. The column "Build%" shows the ratio of the clauses that are built to the total number of learned clauses, which is listed under column "#Nodes" for resolution graph in Table 4.2. As shown in Table 4.3, the checker only needs to construct between 19% to 90% learned clauses to check the proof. Moreover, the instances that take a long time to finish often need a smaller percentage of clauses built to check the proofs. A notable exception to this is longmult12, which is derived from a multiplier. The original circuit contains many xor gates. It is well known that xor gates often require long proofs by resolution.

For both the depth-first and breadth-first approaches, the actual run time to check the proof as well as the peak memory usage are shown under the columns "Time(s)" and "Mem(K)" respectively. Comparing these two approaches, in all test cases, the depth-first approach is faster by a factor of around 2. However, it requires much

Instance Name	Depth-First				Breadth-First	
	Cls Bld	Build%	Time (s)	Mem (K)	Time (s)	Mem (K)
2dlx_cc_mc_ex_bp_f	8105	84.83%	0.84	7860	1.30	4652
bw_large.d	5513	77.21%	1.48	8720	2.44	9920
c5315	27516	54.71%	2.80	18108	5.19	3732
too_largefs3w8v262	56193	61.28%	3.79	26752	5.47	6164
c7552	56603	56.33%	6.16	41420	11.44	5976
5pipe_5_ooo	24910	31.23%	6.60	50044	13.29	17936
barrel9	34624	28.60%	4.85	31456	10.46	6752
longmult12	118615	90.10%	25.87	154288	41.22	7488
9vliw_bp_mc	77296	30.24%	12.78	126752	33.81	17724
6pipe_6_ooo	89695	19.41%	38.52	249468	102.67	40136
6pipe	*	*	*	*	301.98	40248
7pipe	*	*	*	*	645.33	62620

Table 4.3: Number of variables and clauses involved in the proofs

more memory than the breadth-first approach and fails on the two hardest instances because it runs out of memory. In contrast, even though the breadth-first approach is slower compared with the depth-first approach, it is able to finish all benchmarks with a relatively small memory footprint.

It is easy to show that both our proof procedures have linear runtime complexity in the size of the trace file. Table 4.3 shows that the actual time needed to check a proof is always significantly smaller compared with the time needed to derive the actual proof.

4.5 Extracting Unsatisfiable Core

A propositional Boolean formula in Conjunctive Normal Form can be written as[4]:

$$F = C_1 C_2 \dots C_n$$

[4]Part of the work described in this section was first reported in [143].

where $C_i(i = 1 \ldots n)$ are clauses. If the CNF formula is unsatisfiable, then there exists a subset of clauses $\phi \subseteq \{C_i | i = 1 \ldots n\}$ such that the formula formed by the conjunction of the clauses in ϕ is unsatisfiable. The formula obtained by the conjunction of the clauses in ϕ is called an *unsatisfiable core* of the original formula. Such an unsatisfiable core may (but not necessarily needs to) contain a much smaller number of clauses than the original formula.

There are many applications that can benefit from the ability to obtain a *small* unsatisfiable core from an unsatisfiable Boolean formula. In applications such as planning [65], a satisfiable assignment for a SAT instance implies that there exists a viable schedule. Therefore, if a planning instance is proven unfeasible, a small unsatisfiable core can help locate the cause for the infeasibility more quickly. In FPGA routing [97], an unsatisfiable instance implies that the channel is unroutable. By localizing the reasons for the unsatisfiability to the unsatisfiable core, the designer can determine the reasons for the failure of routing more easily. Another application for unsatisfiable core is discussed by McMillan [92], who uses the core extracted from a proof of property by a bounded model checker (using SAT solver) to act as an abstraction for unbounded model checking to prove the property.

The unsatisfiable core problem has been studied by Bruni and Sassano in [14]. The authors discussed a method for finding *minimal unsatisfiable subformula (MUS)* of an unsatisfiable CNF instance. The approach uses an adaptive search procedure to heuristically estimate the *hardness* of the clauses, and tries to find a small subformula consisting of clauses that are deemed *hard* in the original formula. The procedure performs SAT checking on the subformula iteratively, beginning with a very small subformula consisting of a small number of the hardest clauses. If the subformula is satisfiable, it adds more clauses into the subformula. If the search procedure cannot determine the satisfiability of the subformula after certain number of branches, it removes some clauses from it. The process goes on until the subformula can be

shown to be unsatisfiable. In that case the subformula is the unsatisfiable core. The search is adaptive in the sense that the hardness of a clause changes during the SAT checking. Their experimental results show that the procedure is successful in finding small unsatisfiable cores for the instances tested. However, the benchmarks they use are very small instances that are trivial for current state-of-the-art SAT solvers. Moreover, their procedure needs careful tuning of run time parameters (i.e. initial size of the subformula, cutoff for SAT checking, number of clauses added/removed for each iteration) for each formula being processed. Therefore, the procedure is not fully automatic. Their procedure may not scale well on difficult SAT instances generated from real world applications because such an iterative procedure takes too much time to converge.

This section describes another procedure that can extract an unsatisfiable core from an unsatisfiable Boolean formula. This procedure is based on the proof checking procedure discussed in the previous sections. When a SAT solver based on DLL proves that a formula is unsatisfiable, a resolution graph that contains an empty clause can be extracted from the solving process. All clauses that are not in the transitive fan-in cone of the empty clause in the resolution graph are not needed to construct the proof. Deleting them will not invalidate the proof of unsatisfiability. More precisely, the source nodes (original clauses) in the transitive fan-in cone of the empty clause is a *minimal* subset of the original clauses that are needed *for this particular proof*. This subset of clauses is an unsatisfiable core of the original formula.

The resolution graph can be easily recorded as a trace file during the SAT solving process, as described in the previous section. A simple traversal of the graph can tell what clauses are in the transitive fan-in cone of the empty clause, as shown in Figure 4.6. Therefore, if a Boolean formula is proven to be unsatisfiable by a SAT solver, an unsatisfiable core can be generated from the proof. The unsatisfiable core of a formula is by itself an unsatisfiable Boolean formula. Therefore, a SAT solver can

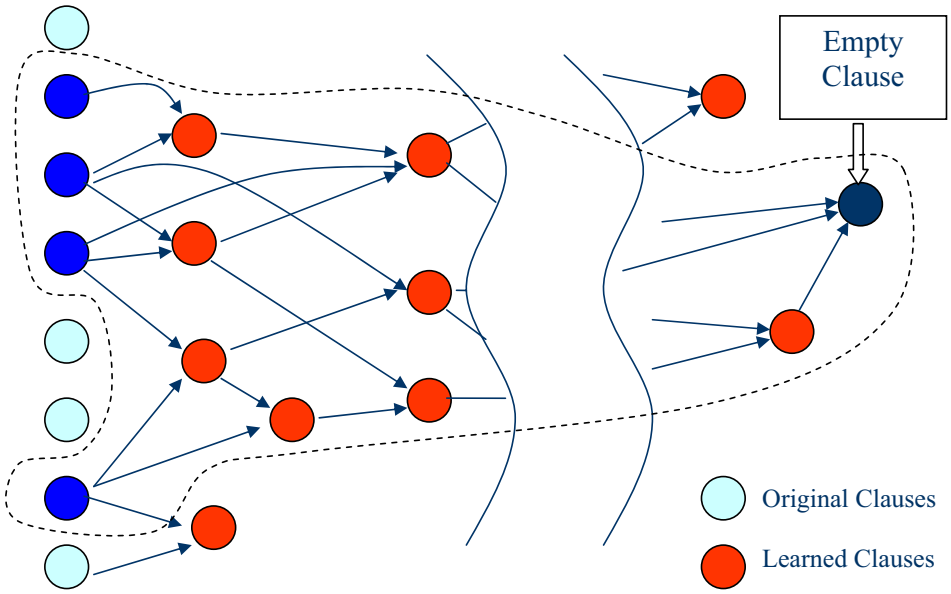


Figure 4.6: Extract unsatisfiable core from the resolution graph

work on it and produce another proof of unsatisfiability and extract an unsatisfiable core from the proof. This process can be carried out iteratively to improve the size of the final result.

The core extracting procedure described is fully automated and efficient enough for extracting unsatisfiable cores from SAT instances generated from real world applications. Because essentially all DLL SAT solvers can be modified to dump out a trace file for the resolution graph, this method can work with any CNF formula as long as some DLL SAT solver can prove that it is unsatisfiable. Moreover, it is possible to tradeoff the quality of the extracted core with the run time for extraction. If the extraction is carried out for just one iteration, an unsatisfiable core can be extracted quickly. It is also possible to run the extraction and solving process for several iterations to improve the quality (i.e. reduce the size) of the extracted unsatisfiable core.

The implementation of these ideas is relatively straight forward. It is a simple

graph traversal algorithm to calculate the fan-in cone of a directed acyclic graph. However, a couple of points should be mentioned here that are important for efficient implementation. The resolution graph is dumped to the hard disk as a trace file by the SAT solver during the solving process. The graph can be very large for hard SAT instances and may not fit in the main memory of the computer. Therefore, the graph traversal has to be carried out on hard disk. Fortunately, the graph stored on the hard disk is already topologically ordered because when a clause is generated by resolution, all its resolve sources must have already been generated. Therefore, if the solver records the resolve sources of clauses to the trace file in the same order as it generates them, the nodes in the file will be topologically ordered. However, to find the transitive fan-in of a node in a graph, the tool needs to traverse the graph in *reverse* topological order. Accessing a hard disk file in the reverse order from how it was stored is very inefficient. Therefore, the extractor needs to reverse it first before the traversal. This can be achieved efficiently by using a memory buffer.

A possible failure for marking the transitive fan-ins of a node in a DAG is that during the traversal, the tool needs storage to indicate whether a yet to be visited node in the graph is in the transitive fan-in or not. A hash table is used for this storage. In theory, the hash table may have the same number of entries as the number of nodes in the unvisited graph, which in turn can be as large as the whole graph. Therefore, theoretically the algorithm may still cause memory overflow for extremely large graphs, because the hash table may be too large. In practice, as shown by the experimental results, the maximum number of hash table entries encountered during the traversal is rarely much larger than the number of clauses in the final unsatisfiable core.

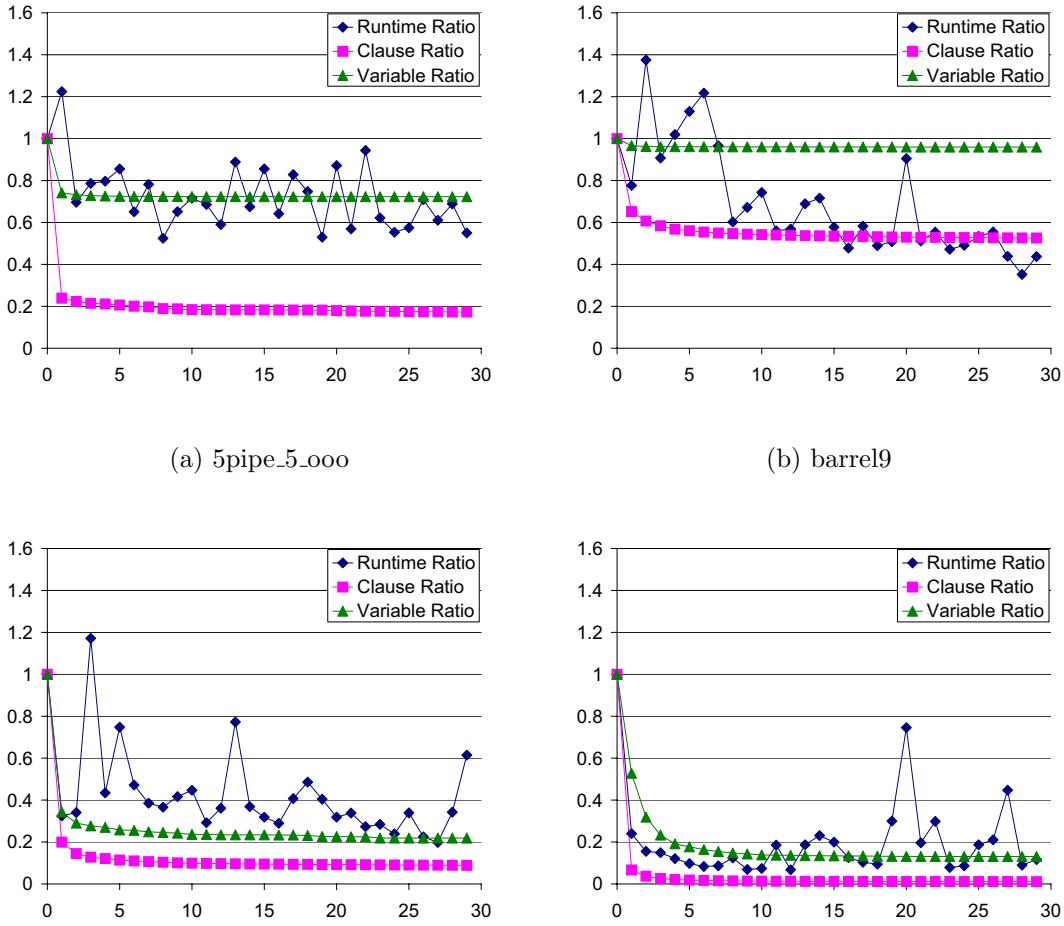
Instance Name	Original		First Iteration				30 Iterations/Fixed Point		
	Num. Clauses	Num. Vars	Num. Clauses	Num. Vars	Hash Size	Extract Time	Num. Clauses	Num. Vars	Num. Itrs.
2dlx_cc_mc_ex_bp_f	41704	4524	11169	3145	11169	0.85	8038	3070	26
bw_large.d	122412	5886	8151	3107	8151	1.54	1364	769	30
c5315	5024	5399	14336	5399	14611	2.64	14289	5399	3
too_largefs3w8v262	50216	2946	10060	2946	11971	2.65	4473	645	30
c7552	20423	7651	19912	7651	23651	5.37	19798	7651	9
5pipe_5_ooo	240892	10113	57515	7494	57521	6.62	41499	7312	30
barrel9	36606	8903	23870	8604	24039	4.66	19238	8543	30
longmult12	18645	5974	10727	4532	26894	21.31	9524	4252	7
9vliw_bp_mc	179492	19148	66458	16737	66458	10.68	36840	16099	30
6pipe_6_ooo	545612	17064	180559	12975	180561	37.14	109369	12308	30
6pipe	394739	15800	126469	13156	126469	99.96	73681	13065	30
7pipe	751118	23910	221070	20188	221079	152.71	133211	20076	30

Table 4.4: Extracting unsatisfiable cores

Table 4.4 shows the statistics for extracting the unsatisfiable core of the SAT instances described in Section 4.2. The instances and statistics of the resolution graphs have already been listed in Table 4.1 and 4.2 in the previous section. The "First Iteration" section shows the runtime and the resulting unsatisfiable core extracted after one iteration of the core extraction procedure. The size of the extracted core as well as run time for the core extraction is shown, along with the maximum hash table size for storing unvisited nodes during the traversal. The columns under "30 Iterations/Fixed Point" report the statistics for running up to 30 iterations of the core extraction procedure. If the "iteration" number is smaller than 30, it means that after certain iterations, all clauses are needed in the proof.

Table 4.4 shows the extracted unsatisfiable core can be much smaller than the original formula in the case of bw_large.d, or almost the same size as in the case of c7552 and c5315. Instance bw_large.d is obtained from SAT planning. A small unsatisfiable core means that the reason for unfeasible scheduling is localized. Instances c7552 and c5315 are obtained from combinational equivalence checking. A large unsatisfiable core means that the two circuits being checked for equivalence do not contain much redundancy. The core extraction is very fast compared with SAT solving. As mentioned previously, the experiment shows that the maximum hash table sizes are barely larger than the number of clauses in the resulting unsatisfiable cores. Comparing the 30 iteration number with first iteration, the core size is often reduced by running the procedure iteratively, but usually the gains for core size reduction are not as large as the first iteration.

Figure 4.7 shows the statistics of the core extractions during the iterations for four representative instances. The three lines represent the zchaff time for the instance, the number of clauses and the number of variables of the extracted cores normalized to the original formula. The X axis is the number of iterations, the Y axis is the ratios. The figure shows that the size of the instance reduces dramatically in the first



(a) 5pipe_5_ooo

(b) barrel9

(c) too_largefs3w8v262

(d) bw_large.d

Figure 4.7: The Core extraction statistics as a ratio of the original. X-Axis: Iterations; Y-Axis: Ratio

Benchmark Instance	**Original #Cls.**	**Extracted #Cls.**	**Minimal #Cls.**	**Clause Ratio**
2dlx_cc_mc_ex_bp_f	41704	8036	7882	1.020
bw_large.d	122412	1352	1225	1.104
too_largefs3w8v262	50416	4464	3765	1.186

Table 4.5: The quality of extracted unsatisfiable cores

iteration, but not by much during the following iterations. The run time for SAT solving on the core is also often reduced compared with the original formula with a trend similar to the reduction in size.

As shown by the previous examples, the procedure described in this section can often extract small unsatisfiable cores from an unsatisfiable formula. However, in this procedure the extracted core is minimal only with respect to the particular proof produced by the SAT solver. There are no guarantees of a *minimal* core (i.e. deleting any clause from the core will render the core satisfiable) or a *minimum* core (i.e. the core extracted is the smallest for the Boolean formula). To estimate the quality of the cores extracted by the procedure, several easier instances are chosen from the benchmarks for the following experiment. Starting from the smallest unsatisfiable core the procedure can generate (i.e. more iterations cannot reduce the core size further), the clauses are deleted one by one until deleting any remaining clause will make the instance satisfiable. Now the resulting formula is a minimal unsatisfiable core for the original formula. The minimal formula is compared with the extracted formula in Table 4.5.

As seen from Table 4.5, for the three benchmarks tested, the cores extracted by the procedure are about 2% to 20% larger than the minimal cores that are extracted by an iterative process. This shows that the procedure is reasonably good for extracting small cores. Notice that the minimal core extraction takes much longer to finish

(essentially it needs to run the SAT solver on the starting instance for iterations as many as the number of clauses of the instance). The author is not aware of any computationally efficient way to find the *minimum* unsatisfiable core, since enumerating all combinations of the clauses is clearly intractable. Therefore, there is no way to compare the size of the extracted cores with the size of the minimum cores of the benchmark instances.

To compare the core extraction procedure proposed in this section with that of Bruni and Sassano [14], some of the experimental data of their work [14] are copied in Tables 4.6 and 4.7 and compared with the results obtained by the procedure proposed in this section. The tables are for two classes of instances from the DIMACS [1] benchmark suite, respectively. The tables show the original number of clauses for each instance. Under "Bruni & Sassno", the size of the extracted unsatisfiable core (in term of number of clauses), run time for extracting the core and the minimality of the core are shown for each instances. The experiments are carried out on a PIII 450 machine. Under "Proposed", only the core sizes and minimality of the cores are shown because the proposed procedure is much faster than the compared method. In fact, the run time for the experiments on all of the instances shown in Table 4.6 and Table 4.7 (including core extractions and the time consuming minimality tests) is less than 10 seconds on a PIII 833 machine. From these two tables it is obvious that the proposed method is at least as good as the iterative method proposed by Bruni and Sassano [14]. In fact, for the relative harder instance class `jnh`, the proposed method usually extracts much smaller cores than the other method.

Zchaff uses a deterministic branching strategy. Therefore, if the input formulas are similar, the solver often produces similar proofs. If random decision strategy is used, the solver is expected to find different proofs and the cores extracted can be even smaller. Moreover, it is possible that the solver may find multiple final conflicting clauses during the solving process. Instead of using just one of them as in

Benchmark	**Original**	**Bruni & Sassano**			**Proposed**	
Instance	**#Cls.**	**#Cls.**	**Time(s)**	**Minimal**	**#Cls.**	**Minimal**
aim-50-1_6-no-1	80	22	0.1	Y	18	Y
aim-50-1_6-no-2	80	32	0.1	Y	32	Y
aim-50-1_6-no-3	80	31	0.1	Y	31	Y
aim-50-1_6-no-4	80	20	0.0	Y	20	Y
aim-50-2_0-no-1	100	22	0.1	Y	22	Y
aim-50-2_0-no-2	100	31	0.3	N	30	Y
aim-50-2_0-no-3	100	28	0.0	Y	28	Y
aim-50-2_0-no-4	100	21	0.3	Y	19	Y
aim-100-1_6-no-1	160	47	1.2	Y	47	Y
aim-100-1_6-no-2	160	54	4.5	N	54	N
aim-100-1_6-no-3	160	57	4.6	N	57	N
aim-100-1_6-no-4	160	48	2.5	Y	48	Y
aim-100-2_0-no-1	200	19	0.5	Y	19	Y
aim-100-2_0-no-2	200	39	0.9	Y	39	Y
aim-100-2_0-no-3	200	27	1.8	Y	25	Y
aim-100-2_0-no-4	200	32	1.6	N	31	Y
aim-200-1_6-no-1	320	55	2.6	Y	55	Y
aim-200-1_6-no-2	320	82	43.0	N	77	N
aim-200-1_6-no-3	320	86	300	N	83	Y
aim-200-1_6-no-4	320	46	2.3	Y	46	Y
aim-200-2_0-no-1	400	54	3.7	N	53	Y
aim-200-2_0-no-2	400	50	3	Y	50	Y
aim-200-2_0-no-3	400	37	0.4	Y	37	Y
aim-200-2_0-no-4	400	42	0.8	Y	42	Y

Table 4.6: Comparison of two core extraction methods: aim

Benchmark	**Original**	**Bruni & Sasano**			**Proposed**	
Instance	**#Cls.**	**#Cls.**	**Time(s)**	**Minimal**	**#Cls.**	**Minimal**
jnh2	850	60	3.2	N	53	Y
jnh3	850	173	29.7	N	152	N
jnh4	850	140	8.2	N	126	N
jnh5	850	125	7.7	N	81	Y
jnh6	850	159	22.9	N	141	N
jnh8	850	91	0.6	N	88	N
jnh9	850	118	1.0	N	93	N
jnh10	850	161	0.1	N	66	Y
jnh11	850	129	19.0	N	127	N
jnh13	850	106	0.1	N	65	Y
jnh14	850	124	0.5	N	80	N
jnh15	850	140	1.4	N	110	N
jnh16	850	321	55.8	N	320	N
jnh18	850	168	40.6	N	154	N
jnh19	850	122	7.4	N	111	N
jnh20	850	120	0.7	N	104	N

Table 4.7: Comparison of two core extraction methods: jnh

the experiments, it is possible to extract smaller unsatisfiable cores by examining the resolution graphs corresponding to each of them.

4.6 Related Work

As mentioned before, a checking process for a DLL based SAT solver have been discussed by Van Gelder [130], but the procedure may not be easily adapted for modern SAT solvers because the solver discussed is based on flipping decision variables instead of asserting clauses. Besides that work, there are two contemporary works along the same line of thought discussed in this chapter. These are now discussed briefly.

Goldberg and Novikov [99] described a method to check a DLL solver's proof by examining the learned clauses. As mentioned in Chapter 2, a conflict clause corresponds to a cut in the implication graph. Therefore, if a learned clause is valid, setting all literals in it to 0 must lead to conflict in the formula. They also described how to extract an unsatisfiable core from the proof by a traversal similar to the depth-first approach described in this chapter. However, this work suffers fromthe memory blowup problem for long proofs, just like the depth-first approach described in this chapter. Moreover, they did not describe the iterative core extraction strategy.

McMillan and Amla [92] described a method to extract an unsatisfiable core from a SAT proof from a bounded model checking problem using resolution graph, and use the core to help with automatic abstraction for unbounded model checking. The paper did not mention anything about checking the actual proof. Like the previous work, the core extraction procedure is also not carried out iteratively.

4.7 Chapter Summary

This chapter describes the concept of a resolution graph for an unsatisfiable proof of a Boolean formula. The method to record this graph as a trace file during the SAT solving process is discussed. A third party checker can check the recorded trace and validate the unsatisfiability of the CNF formula, thus providing a certificate for the unsatisfiable proof. An unsatisfiable core extractor can extract a small unsatisfiable subformula from the original CNF formula by utilizing the information of the resolution graph trace. The extracted unsatisfiable core is useful for many applications.

As for future work, in the SAT checker part, only DLL based SAT solvers with learning and non-chronological backtracking are discussed. The SAT solver is required to use only unit implication (BCP) for deduction. Even though this is the most widely used algorithm for SAT solving, it is desirable to extend the certification process so that it can work with more reasoning mechanisms.

In the unsatisfiable core extraction part, currently the SAT solvers are not aware of the core extraction process that works on the resolution graph it produces. It would be desirable to make the SAT solver produce a proof that contains a small number of original clauses in order to get a smaller unsatisfiable core by the core extractor. Moreover, the core extracted by the procedure is neither minimal nor minimum with regard to the original formula. It would be desirable to generate minimum or at least minimal cores given an unsatisfiable formula.

The work described in this chapter, i.e. the SAT solver validation and unsatisfiable core extraction, serve the purpose of facilitating the SAT solver as a reasoning engine for real world applications. There are many other features that are desirable for SAT solvers such as monitoring the progress of the solving process, estimating the difficulty of a SAT instance without actually solving it, etc. More research effort is needed on these aspects of SAT solving.

Chapter 5

Learning and Non-Chronological Backtracking in a QBF Solver

In the last few chapters, the techniques used for solving the Boolean Satisfiability (SAT) problem are discussed. As shown by previous results, the techniques of learning and non-chronological backtracking [88, 64] are key to the success of current SAT solvers. In this chapter, the same ideas are applied to another important problem, namely checking the satisfiability of a quantified Boolean formula. A quantified Boolean formula is a propositional Boolean formula with existential and universal *quantifiers* preceding it. Given a quantified Boolean formula, the question whether it is satisfiable (i.e. evaluates to 1) is called the Quantified Boolean Satisfiability problem (QBF). As will be seen in this chapter, there are several similarities between QBF and SAT. Just like the case of SAT solving, learning and non-chronological backtracking can also greatly help QBF solving.

This chapter is organized as follows. Section 5.1 describes the QBF problem and its applications in EDA. Section 5.2 discusses the semantics of a quantified Boolean formula and shows how to apply the DLL algorithm to solve the QBF problem. Section 5.3 discusses how to incorporate conflict-driven learning and non-chronological backtracking in a DLL based QBF solver. Section 5.4 discusses satisfaction-driven

learning and non-chronological backtracking. Some experimental results are shown in Section 5.5. Section 5.6 discusses some related developments. Finally the chapter is summarized in Section 5.7.

5.1 The QBF Problem

The *existential* quantification of function f with respect to variable x is defined as:

$$\exists x f = f(x = 1) + f(x = 0) \tag{5.1}$$

Where $f(x = 1)$ is obtained by setting variable x to the value 1 in f. Similarly, $f(x = 0)$ is obtained by setting x to 0.

The *universal* quantification of function f with respect to variable x is defined as:

$$\forall x f = f(x = 1) \cdot f(x = 0) \tag{5.2}$$

Given a function f of n variables x_i $(i = 1 \ldots n)$, $\exists x_k f$ and $\forall x_k f$, $(k = 1 \ldots n)$, are functions with $n - 1$ variables x_i $(i = 1 \ldots n, i \neq k)$. Given a variable assignment $x_1, x_2, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n$, the function $\exists x_k f$ is true as long as there exists one assignment (0 *or* 1) of x_k that can make f true under the variable assignment. The function $\forall x_k f$ is true if and only if both assignments (0 *and* 1) of x_k make the function f true under the variable assignment. Obviously, if f is not dependent on variable x, then

$$\forall x f = f \tag{5.3}$$

$$\exists x f = f \tag{5.4}$$

Some well-known properties of quantifications are listed here.

$$\exists x \exists y f = \exists y \exists x f \tag{5.5}$$

$$\exists x (f + g) = \exists x f + \exists x g \tag{5.6}$$

$$\forall x \forall y f = \forall y \forall x f \tag{5.7}$$

$$\forall x (f \cdot g) = \forall x f \cdot \forall x g \tag{5.8}$$

Now that the notion of quantification is defined, a quantified Boolean formula can be defined as follow:

Definition 1: *Quantified Boolean Formula*

1. *If f is a propositional formula, it is also a quantified Boolean formula.*
2. *If f is a quantified Boolean formula, and x is a Boolean variable, then both $\exists x f$ and $\forall x f$ are quantified Boolean formulas;*
3. *If f and g are quantified Boolean formulas, then **not**(f), f **and** g, and f **or** g are quantified Boolean formulas;*

As in the propositional case, this chapter uses "+" to represent logical **or** and "·", or just concatenation to represent logical **and**. If f and g are two quantified Boolean formulas, and variable x does not appear in g , then:

$$(\exists x f) + g = \exists x (f + g) \tag{5.9}$$

$$(\exists x f) \cdot g = \exists x (f \cdot g) \tag{5.10}$$

$$(\forall x f) + g = \forall x (f + g) \tag{5.11}$$

$$(\forall x f) \cdot g = \forall x (f \cdot g) \tag{5.12}$$

Equations 5.9 and 5.12 are easy to prove. Equations 5.10 and 5.11 can be proven using Shannon's expansion. Shannon's expansion states that for any Boolean function f:

$$f = x \cdot f(x = 1) + x' \cdot f(x = 0) \tag{5.13}$$

Therefore the right hand side of 5.10 is:

$$\begin{aligned} & \exists x(f \cdot g) && (5.14)\\ = \; & \exists x((xf(x=1) + x'f(x=0)) \cdot g) && (5.15)\\ = \; & \exists x(x \cdot g \cdot f(x=1) + x' \cdot g \cdot f(x=0)) && (5.16)\\ = \; & g \cdot f(x=1) + g \cdot f(x=0) && (5.17) \end{aligned}$$

And the left hand side of 5.10 is:

$$\begin{aligned} & (\exists x f) \cdot g && (5.18)\\ = \; & (f(x=1) + f(x=0)) \cdot g && (5.19)\\ = \; & g \cdot f(x=1) + g \cdot f(x=0) && (5.20) \end{aligned}$$

Therefore, LHS=RHS. 5.11 can be proven in a similar way.

Moreover, if notation $f_{x \to y}$ is used to represent the function resulting from replacement of (renaming) all occurrences of variable x in function f with variable y, while y originally does not appear in f, then:

$$\forall x f = \forall y f_{x \to y} \tag{5.21}$$

$$\exists x f = \exists y f_{x \to y} \tag{5.22}$$

By using 5.9 to 5.12, 5.21 and 5.22, it is possible to move the quantifiers of a quantified Boolean formula to the front of the propositional part by renaming some of the variables. An example of such a transformation is shown in the following:

$$\begin{aligned} & (\exists x \forall y F(x,y,z)) \cdot (\forall x G(x,y,z) + \exists z H(a,b,z)) && (5.23)\\ = \; & (\exists x_1 \forall y_1 F(x_1,y_1,z)) \cdot (\forall x_2 G(x_2,y,z) + \exists z_1 H(a,b,z_1)) && (5.24)\\ = \; & (\exists x_1 \forall y_1 F(x_1,y_1,z)) \cdot (\forall x_2 (G(x_2,y,z) + \exists z_1 H(a,b,z_1))) && (5.25) \end{aligned}$$

$$= (\exists x_1 \forall y_1 F(x_1, y_1, z)) \cdot (\forall x_2 (\exists z_1 (G(x_2, y, z) + H(a, b, z_1)))) \quad (5.26)$$

$$= (\exists x_1 \forall y_1 F(x_1, y_1, z)) \cdot (\forall x_2 \exists z_1 (G(x_2, y, z) + H(a, b, z_1))) \quad (5.27)$$

$$= (\exists x_1 \forall y_1 (F(x_1, y_1, z) \cdot (\forall x_2 \exists z_1 (G(x_2, y, z) + H(a, b, z_1)))) \quad (5.28)$$

$$= \exists x_1 \forall y_1 (\forall x_2 \exists z_1 ((F(x_1, y_1, z)(G(x_2, y, z) + H(a, b, z_1))))) \quad (5.29)$$

$$= \exists x_1 \forall y_1 \forall x_2 \exists z_1 (F(x_1, y_1, z)(G(x_2, y, z) + H(a, b, z_1))) \quad (5.30)$$

Therefore, it is possible to write a quantified Boolean formula in the form:

$$Q_1 x_1 Q_2 x_2 \ldots Q_n x_n \varphi \quad (5.31)$$

Where φ is a propositional formula involving variables x_i, $(i = 1 \ldots n)$. Each Q_i is either an existential quantifier $\exists$ or an universal quantifier $\forall$. Because of Equations 5.5 and 5.7, it is always possible to group the quantified variables into disjoint sets where each set consists of adjacent variables with the same type of quantifier. Therefore, 5.31 can be rewritten in the following form:

$$Q_1 X_1 Q_2 X_2 \ldots Q_m X_m \varphi \quad (5.32)$$

Here, X_i's are mutually disjoint sets of variables. It is easy to observe that such a transformation takes linear time in the size of the original formula. A variable is called existential or universal according to the quantifier of its quantification set. In the future, for convenience of discussion, each variable set will be assigned a *quantification level*. The variables belonging to the outermost quantification set have quantification level 1, and so on.

Given a quantified Boolean formula in the form of 5.32, it is possible to get rid of the quantifiers by expanding the formula following the definition of the quantifications as in 5.1 and 5.2. The result is a propositional formula with the number of variables equal to the number of variables in φ minus n, where n is the total number of variables in the set X_i. However, such an expansion has an obvious exponential space (and run time) complexity.

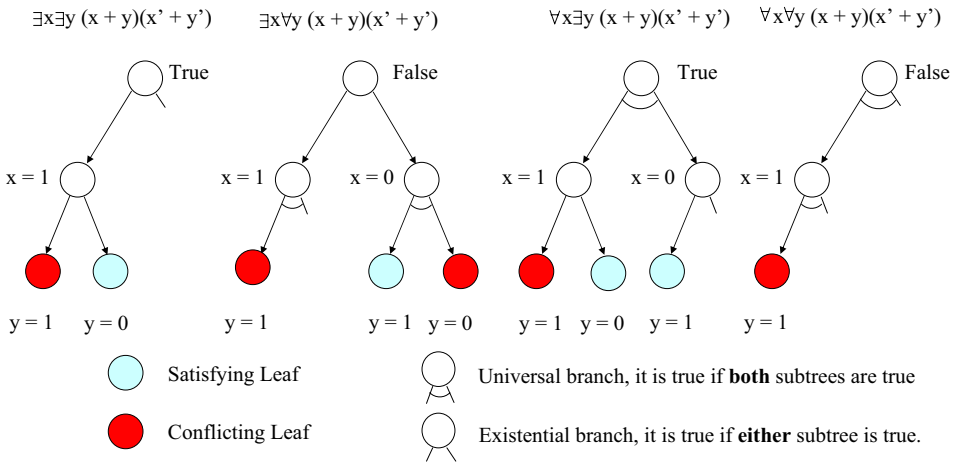


Figure 5.1: Some examples of QBF Instances

If *all* the variables appearing in φ are required to appear in one of the quantification sets, then the quantified Boolean formula must evaluate to a constant Boolean value. Given such a quantified Boolean formula, the *QBF problem* asks whether it evaluates to *true* (satisfiable) or *false* (unsatisfiable). In the rest of the chapter, unless mentioned otherwise, only quantified Boolean formulas with all variables appearing in the quantification sets will be discussed. Also, both the quantified Boolean formula and the decision problem may be referred to as QBF when no confusion may result. Some examples of QBF instances are shown in Figure 5.1. The examples also illustrate how quantifiers affect the outcomes of the QBF problems.

The QBF problem is shown to be PSPACE-Complete [39]. A PSPACE-Complete problem is a decision problem that can be solved in polynomial space (the *PSPACE* part). Moreover, any decision problem that can be solved in polynomial space can be reduced to it in polynomial time (the *complete* part). The Boolean Satisfiability problem (SAT) can be regarded as a special case of the QBF problem with only existential variables. SAT is NP-Complete and lies lower in the complexity hierarchy than QBF.

In practice, QBF can be used to solve many practical problems ranging from AI planning [108] to sequential circuit verification [11, 118]. For example, Biere *et al.* [11] showed that the problem of determining if a certain number equals the diameter of the state space for a finite state system can be reduced to a QBF instance. The diameter of the state space is the minimum number of transitions needed to visit an arbitrary reachable state from a given initial state. If the diameter of the state space is known, bounded model checking [10] can be made complete in many interesting situations (i.e. for property checking). Completeness here referred to the ability to fully prove the property even though bounded model checking only check the property for limited time frames. Because of these applications, there is interest in efficient QBF solving.

5.2 Solving QBF using DLL search: Previous Works

Research on QBF solvers started in the mid 1990s. A resolution-based algorithm [18] was proposed and shown to be complete and sound in 1995. Plaisted *et al.* [102] proposed a QBF evaluation algorithm that eliminates variables from inside out (i.e. from higher quantification order variables to the lower quantification order variables).

Researchers have also investigated QBF solvers based on the Davis Logemann Loveland [31] search procedure (e.g. [21, 107, 42]). QBF solvers based on DLL search do not blow up in space. This section discusses this approach in more detail. Section 5.2.1 discusses the semantic tree of a QBF problem. The DLL procedure is formalized for QBF solving and the similarities and differences of QBF solving with SAT solving are discussed in Section 5.2.2. Section 5.2.3 discusses some of the deduction rules that can be used in a QBF DLL search algorithm to prune the search space.

5.2.1 The Semantic Tree of QBF

The QBF problem is best illustrated with the semantic tree of a quantified Boolean formula [73]. The semantic tree of a QBF is a binary tree. A node in the tree represents a quantified Boolean formula. Notice in the discussion, a QBF evaluates to a constant value. Therefore, a formula and the value of the QBF are referred to interchangeably. The root node of a semantic tree represents the original QBF. As an example, a semantic tree for the QBF $\exists x_1 \forall y_1 y_2 \exists z_1 f(x_1, y_1, y_2, z_1)$ is shown in Figure 5.2. In a semantic tree, each non-leaf node is associated with a variable, the two children of the node corresponds to the variable set to 0 and 1 respectively. In effect, an internal node in the tree represents a QBF that contains one less variable than its parent node. The child node's propositional part is the parent node's propositional part with the variable set to a constant value. For example, in Figure 5.2 the root node is associated with variable x_1. The left child of the root node in represents $\exists x_1 \forall y_1 y_2 \exists z_1 (f(x_1 = 1, y_1, y_2, z_1))$, which is equal to $\forall y_1 y_2 \exists z_1 (f(x_1 = 1, y_1, y_2, z_1))$ because $f(x_1 = 1, y_1, y_2, z_1)$ does not depend on x_1 (see Equation 5.4).

The value of a leaf node in the semantic tree of a QBF is equal to a full assignment of the variables for the propositional part of the QBF. There are 2^n different full variable assignments for a QBF with n variables, each of which is represented by one leaf in the semantic tree. For example, the leftmost leaf node in Figure 5.2 corresponds to $f(x_1 = 1, y_1 = 1, y_2 = 1, z_1 = 1)$.

In a semantic tree, the internal nodes in the tree are ordered according to the quantification levels of the variables associated with the nodes. A node must lie below any nodes whose associated variables have a lower quantification level than the associated variable for that node. For example, because z_1 has the highest quantification level (3), it's nodes must lie below nodes of quantification level 1 (i.e. nodes for x_1) and level 2 (i.e. nodes for y_1 and y_2). Within the same quantification level, the nodes can be ordered arbitrarily. For example, in Figure 5.2 the left half nodes of

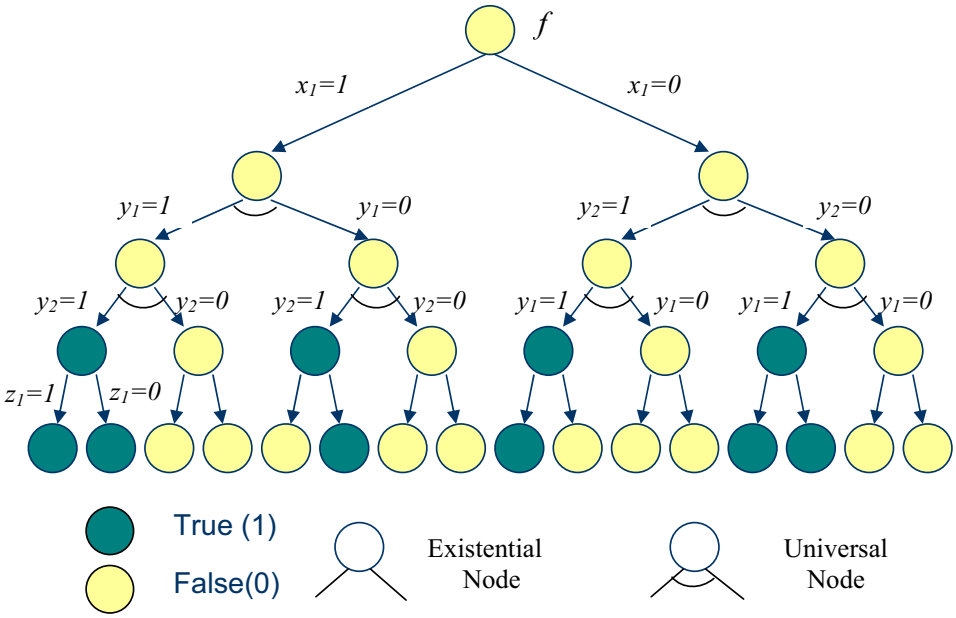


Figure 5.2: Semantic tree for $\exists x_1 \forall y_1 y_2 \exists z_1 f(x_1, y_1, y_2, z_1)$

y_1 are above nodes of y_2 while in the right half nodes of y_1 are below nodes of y_2. In a semantic tree, an internal node is either *universal* or *existential* depending on the variable associated with it. In Figure 5.2 a universal node has a arc below it while an existential node has not.

The nice property of a semantic tree is that it is easy to determine the value of a node if both values of its children nodes are known. A universal node is *true* if and only if both of its children are *true*. An existential node is true if either of its children is *true*. The QBF problem essentially asks the question whether the root node is *true* or *false*. Therefore, if the semantic tree can be constructed, the value of the root node can be determined easily. In the rest of the chapter, a *true* node is called a *satisfying* node, and the partial variable assignment corresponding to that node is called a *satisfying assignment*. Similarly, a *false* node is called a *conflicting* node, and the corresponding partial variable assignment is called a *conflicting assignment* or simply a *conflict*.

Given a particular QBF instance, its semantic tree is not unique. Because variables with the same quantification order can be ordered arbitrarily in the tree, it is possible to have many different trees with different order of variables within the same quantification level. However, it is easy to prove that the valuation for the topmost nodes in each quantification level are the same for different semantic trees. In particular the root nodes have the same value for all semantic trees.

To solve a QBF problem, the most obvious approach is to just naïvely construct and search the whole semantic tree. However, this is not practically feasible because of the exponential number of leaves. Therefore, some pruning techniques are needed in order to reduce the search space. A valid QBF solver, when applied to a QBF instance, should return a value consistent with the value of the semantic tree of the QBF. Therefore, the pruning techniques applied to a solving process must preserve the values of the nodes in the semantic tree.

5.2.2 DLL Search for QBF

From the previous section it is easy to see that the semantic tree of a QBF problem is very similar to the search tree of a SAT problem (see Figure 2.1). Therefore, a search procedure can also be used to solve the QBF problem. A search-based algorithm essentially searches the entire Boolean space B^n to determine the value of a given formula with n variables. Search algorithms are not necessarily limited to depth-first search (e.g. DLL) or breadth-first search (e.g. Stålmarck's algorithm), nor do they have to search according to some semantic trees. As long as the value of the 2^n leaves of the tree are known, the valuation of the root node can always be determined.

However, a search based algorithm needs to reconstruct the semantic tree in order to determine the value of the root node. Notice that a nice property of a QBF semantic tree is that whenever the value of a node is known, the values of its children are no longer needed for the purpose of constructing the value of the root node. Therefore,

if the search is a DLL like depth-first search on a semantic tree, at most the values of n nodes need to be kept for a formula with n variables. Other search methods (or a depth-first search on a non-semantic tree) may incur exponential memory requirement for the solver. For example, a breadth-first search on the tree needs to store the value of 2^n leaves before the values of the nodes at depth $n-1$ can be determined.

The most widely used algorithm for QBF solving is the application of the DLL algorithm to a semantic tree of a QBF instance. To make the search tree the same as the semantic tree, the solver needs to branch according to the quantification order. As mentioned before, such a search algorithm needs to keep only a polynomial number of values for of nodes. Therefore, it can execute in polynomial space (PSPACE). The recursive procedure for a QBF solver is shown in Figure 5.3. It is essentially a depth-first traversal of the semantic tree for a QBF. In the future, when DLL is mentioned in a QBF context, it will always be assumed that the search is on a semantic tree of the QBF.

The recursive QBF DLL algorithm in Figure 5.3 is very similar to the DLL algorithm for SAT shown in Figure 2.2. There are some important differences:

- **The branching order**

  As mentioned earlier, in QBF, the variable branching needs to observe the quantification order, i.e., the solver cannot branch on free variables with higher quantification order unless all variables with lower quantification order are assigned a value. If there is more than one free variable at the lowest quantification level, the variable choice among them can be arbitrary. Essentially, among all the semantic trees of the QBF instance, the solver can arbitrarily select one to traverse.

- **Existential and universal variables**

```
DLL_QBF(formula, assignment) {
    necessary = deduction(formula, assignment);
    new_asgnmnt = union(necessary, assignment);
    if (is_satisfied(formula, new_asgnmnt))
        return SATISFIABLE;
    else
    {
        if (is_conflicting(formula, new_asgnmnt))
            return CONFLICT;
    }
    var = choose_free_var_by_q_order(formula, new_asgnmnt);
    asgn1 = union(new_asgnmnt, assign(var, 1));
    result1 = DLL(formula, asgn1);
    asgn2 = union (new_asgnmnt, assign(var, 0));
    if (is_universal(var)) {
        if (result1 == CONFLICT)
            return result1;
        else
            return DLL_QBF(formula, asgn2));
    }
    else {
        if (result1 == SATISFIABLE)
            return result1;
        else
            return DLL_QBF(formula, asgn2));
    }
}
```

Figure 5.3: The recursive description of the DLL algorithm for QBF solving

Unlike SAT, in which all variables are existential, in QBF existential and universal variables have different semantics and need to be treated differently. For universal variables, a node's value is the logical **and** of its two children, while for existential variables, a node's value is the logical **or** of its two children.

- **Deduction rules**

  In QBF, techniques to prune the search space, i.e., the deduction rules, are different from the SAT case. The functions for deducing the consequence of an assignment, for determining conflicts, and for determining satisfaction, viz. `deduction()`, `is_conflicting()` and `is_satisfied()` respectively, have different meanings from their counterparts in Figure 2.2. Similar to SAT, the algorithm shown in Figure 5.5 is correct regardless of the actual rules implemented in deduction and branching as long as the rules are valid (i.e., will not report a formula to be `SATISFIABLE` when it is actually `CONFLICTING`). However, just like SAT, certain rules are more efficient than others for QBF solving.

The algorithm described in Figure 5.5 does not limit the QBF to any particular form. However, just like SAT, usually the propositional part of the input formula is required to be in Conjunctive Normal Form (CNF). This is because there are some simple rules for QBF in CNF that can be used to prune the search space with relatively small computational effort.

5.2.3 Deduction Rules

QBF in CNF has some nice properties that can be used as deduction rules to prune the search space. Many of these rules are described by Cadoli *et al.* [21], who proposed the first DLL solver for QBF. This section briefly reviews some of these rules.

If the propositional part of the QBF is in Conjunctive Normal Form (CNF),

Equation 5.32 can be rewritten as:

$$Q_1 X_1 Q_2 X_2 \ldots Q_n X_n C_1 C_2 \ldots C_m \tag{5.33}$$

Here C_i's are clauses. In the following, a QBF in form 5.33 is called a QBF in Conjunctive Normal Form (CNF). The rest of the chapter will use $E(C)$ and $U(C)$ to represent the set of existential and universal literals respectively in the clause C. Also, e, e_i $(i = 1, 2, \ldots, n)$ will be used to represent existential literals and u, u_i $(i = 1, 2, \ldots, n)$ will be used to represent universal literals. $V(l)$ will be used to denote the value of a literal l, and $L(l)$ will be used to denote the quantification level of the variable corresponding to literal l.

The following are some of the rules that can be used for DLL search for QBF in CNF.

Rule 1: ***Unit Implication Rule for Non-Tautology Clauses.***

For a QBF formula F in the form of 5.33, if at a certain node in the search tree, a non-tautology clause C has a literal e such that:

1. $e \in E(C), V(e) = X$. *For any* $e_1 \in E(C), e_1 \neq e : V(e_1) = 0$;

2. $\forall u \in U(C), V(u) \neq 1$. *If* $V(u) = X$, *then* $L(u) > L(e)$

Then the value of the current node is the same as the QBF formula resulting from replacing the literal e with value 1 in the formula that the current node represents .

When a non-tautological clause satisfies the two conditions listed in Rule 1, the clause is said to be a *unit clause*, and the literal e is the *unit literal*. Notice that the unit literal of a unit clause is always an existential literal. By Rule 1, when a unit clause exists, there is no need to explore the search space where the unit literal evaluate to 0 because that space will not affect the outcome of the current node. The solver can directly assign the unit literal e with value 1. The operation of assigning

the unit literal of a unit clause with value 1 is called *implication*, and the unit clause is said to be the *antecedent* of the unit literal.

Rule 2: *Conflicting Rule for Non-Tautology Clauses.*

For a QBF F in the form of 5.33, if at a certain node in the search tree, there exists a non-tautology clause C, such that $\forall e \in E(C), V(e) = 0$, and $\forall u \in U(C), V(u) \neq 1$, then the current node evaluates to 0.

When a non-tautological clause satisfies the condition of Rule 2, the clause is said to be *conflicting*. If there exists a conflicting clause in the clause database, then it is futile to search the rest of the subtree rooted at that node. Therefore, the solver can label current node as 0 and backtrack immediately.

Rule 3: *Pure Literal Rule for Existential Variables.*

For a QBF F in the form of 5.33, if at a certain node in the search tree, there exists an existential *literal e such that its complement e' only appears in clauses that contain value 1 literals or does not appear in the formula at all, then the value of the current node is the same as the QBF formula resulting from replacing the literal e with value 1 in the formula that the current node represents.*

Rule 4: *Pure Literal Rule for Universal Variables.*

For a QBF F in the form of 5.33, if at a certain node in the search tree, there exists a universal *literal u such that its complement u' only appears in clauses that contain value 1 literals or does not appear in the formula, then the value of the current node is the same as the QBF formula resulting from replacing the literal u with value 0 in the formula that current node represents.*

Rules 3 and 4 are collectively called pure literal rules or monotonic literal rules. To understand these two rules, it is helpful to think of a QBF instance as a *game*.

In a game, there are two opponents, each with opposing *goals*. The opponents try to achieve their goals by performing *moves*. The moves must obey some *rules*. In the QBF case, the opponents are universal variables and existential variables. The goal of the universal variables is to try and falsify the formula, while the goal of the existential variables is to make the formula evaluate to true. The move for each opponent is to select values for the variables to be assigned. The rule is that they must select values in an order consistent with the quantification levels. Only after all variables that have lower quantification levels have selected their values, can a variable with a higher quantification level select its value. Using this notion of a game, it is easy to understand the pure literal rules. As universal literals want to falsify the formula, if there exists a pure universal literal, then it will always try to assign the literal to be 0 because that maximizes the chances that the formula will be *false*. A similar reasoning applies for the pure existential literals.

These four rules can be implemented in functions `deduction()` (Rules 1, 3 and 4) and `is_conflicting()` (Rule 2) in Figure 5.3. The pseudo-code for these functions is shown in Figure 5.4. When the QBF formula is in CNF, the function `is_satisfied()` is no different than the one used in SAT algorithms in Figure 2.2. One method is to test all clauses and see if all of them are satisfied or not. If they are all satisfied, then the formula is satisfied. The pseudo-code corresponding to this method is shown here. An alternative way is to test if all free variables are assigned a value or not. If that is the case and deduction finds no conflict in the clause database, then the formula is satisfied.

The above four rules (together with some other rules) were first introduced and proved Cadoli *et al.* [21]. Notice that both Rules 1 and 2 require the clause to not be a tautology. Their proofs are not applicable when the clause is a tautology. Given a QBF in CNF, it is easy to make the original inputs free of tautology clauses by deleting all the tautology clauses from the formula. Therefore, the non-tautology requirement

```
deduction(formula, assignment) {
    needed_assignment = EMPTY_SET;
    while(exist_unit_clause() || exist_pure_literal())
    (
        if (exist_unit_clause())
        (
            cl = get_unit_clause();
            lit = find_unit_literal_from_clause(cl);
            needed_assignment.insert(lit);
            assign(lit, 1);
        }
        else
        {
            lit = get_pure_literal();
            needed_assignment.insert{lit);
            if (is_universal(lit))
                assign(lit, 0);
            else
                assign(lit, 1);
        }
    }
    return needed_assignment
}

is_conflicting(formula, assignment) {
    if (exist_conflicting_clause())
        return true;
    return false;
}

is_satisfied(formula, assignment) {
    if (all_clauses_have_at_least_one_value_1_literal())
        return true;
    return false;
}
```

Figure 5.4: The functions for deduction in QBF on a CNF database

for the clauses is not an issue here. However, when learning is introduced in the next section, tautology clauses may be introduced during the search process. In that case, their proofs are not valid. However, these rules are still valid as will be shown by the next section.

5.3 Conflict-Driven Learning

In the Boolean Satisfiability case, conflict-driven learning and non-chronological backtracking are key to the success of SAT solvers[1]. They are very efficient in pruning the search space for structured instances. However, the QBF algorithm described in Figure 5.3 does not employ learning and non-chronological backtracking. This section describes how a QBF solver can learn and backtrack non-chronologically from a conflicting leaf. In a search tree for a QBF problem, the solver may also encounter satisfying leafs. Learning and non-chronological backtracking for satisfying leaves will be discussed in the next section.

5.3.1 The Top-Level Algorithm

Similar to the SAT case shown in Figure 2.5, the recursive DLL algorithm can be rewritten in an iterative formulation, as shown in Figure 5.5. Just as in the SAT case, this formulation enables the solver to incorporate learning and non-chronological backtracking, which is not possible in a recursive formulation of the algorithm.

For the same reasons as the recursive algorithm, the decision procedure in Figure 5.5 (i.e. function `decide_next_branch()`) needs to obey the quantification order. Just like the SAT case, decision variables have decision levels and implied variables assume the same decision level as the decision variable that transitively causes the implication. When the formula is conflicting under the current assignment, the solver

[1]Part of the work described in this section was first reported in ICCAD 2002 [140].

analyzes the conflict and backtracks. Unlike the SAT case, when a satisfying assignment is encountered, the solver still needs to backtrack because of the universal variables.

The backtracking level is decided by function `analyze_conflict()` and function `analyze_satisfiability()`. The naïve way to determine the backtrack level is as follows. Each decision variable has a flag associated with it. The value of the flag is either *flipped* or *unflipped*. When a decision is made, the flag assumes the initial value of *unflipped*. Function `analyze_conflict()` searches for the unflipped *existential* decision variable with the highest decision level, flips it (i.e. makes it assume the opposite phase, and marks it flipped), and returns that decision level. Similarly, Function `analyze_satisfiability()` searches for the unflipped *universal* decision variable with the highest decision level, flips it, and returns the decision level. When no such decision variable exists, both routines return -1.

When using the naïve backtracking method, the iterative algorithm described in Figure 5.5 is essentially the same as the recursive algorithm described in Figure 5.3. Many QBF solvers (e.g. [21, 108]) use this naïve backtracking method.

Similar to SAT, it is possible do better than the naïve backtracking by introducing learning and non-chronological backtracking. This section describes how to accomplish this when a conflicting leaf is encountered (i.e. conflict-driven learning and non-chronological backtracking). The next section will describe how to accomplish this when a satisfying leaf is encountered (i.e. satisfaction-driven learning and non-chronological backtracking).

5.3.2 Conflict Analysis in a QBF Solver

Conflict-driven learning and non-chronological backtracking in a SAT solver was independently proposed by Silva and Sakallah [88], and Bayardo and Schrag [64]. This technique is very effective in pruning the search space for structured SAT instances

```
DLL_QBF_iterative() {
    status = preprocess();
    if (status!=UNKNOWN)
        return status;
    while(1) {
        decide_next_branch();
        while (true)
        {
            status = deduce();
            if (status == CONFLICT)
            {
                blevel = analyze_conflict();
                if (blevel < 0)
                    return UNSATISFIABLE;
                else
                    backtrack(blevel);
            }
            else if (status == SATISFIABLE)
            {
                blevel = analyze_satisfiability();
                if (blevel < 0)
                    return SATISFIABLE;
                else
                    backtrack(blevel);
            }
            else break;
        }
    }
}
```

Figure 5.5: The iterative description of the DLL algorithm for QBF solving

```
analyze_conflict(){
    if (current_decision_level()==0)
        return -1;
    cl = find_conflicting_clause();
    do {
        lit = choose_literal(cl);
        var = variable_of_literal( lit );
        ante = antecedent( var );
        cl = resolve(cl, ante, var);
    } while (!stop_criterion_met(cl));
    add_clause_to_database(cl);
    back_dl = clause_asserting_level(cl);
    return back_dl;
}
```

Figure 5.6: Conflict analysis in a QBF solver

and can greatly improve the efficiency of SAT solvers. This section shows how to incorporate this idea in a QBF solver.

In a SAT solver, the learning and backtracking scheme is always conflict-directed because whenever a conflict is found, the solver needs to backtrack and find a new space to search. In QBF solvers, the learning and backtracking can either be conflict-directed or satisfaction-directed. When the solver encounters a conflict, it needs to backtrack; when it finds a satisfying assignment, because of the universal variables, it also needs to backtrack. This section assumes that the satisfiability analysis is still carried out in a naïve way while concentrating on the conflicting case. The top-level pseudo-code for conflict-driven learning and backtracking, performed by function `analyze_conflicit()` in Figure 5.5 for a QBF solver, is shown in Figure 5.6.

Comparing Figure 5.6 to the procedure for conflict analysis in the SAT solver chaff (shown in Figure 4.2), it is easy to see that they are exactly the same. At the beginning, the function checks whether the current decision level is already 0. If that is the

case, the function returns -1, indicating that there is no way to resolve the conflict and the formula is unsatisfiable. Otherwise, the solver determines the cause of the conflict. Iteratively, the procedure resolves the conflicting clause with the antecedent clause of a variable that appears in the conflicting clause. Function `choose_literal()` always chooses an implied *existential* literal from the input clause in reverse chronological order (i.e., the variable assigned last is chosen first). Function resolve `(cl1,cl2,var)` returns a clause that has all the literals appearing in $cl1$ and $cl2$ except for the literals corresponding to var. If the generated clause meets some predefined stopping criterion, the iteration will stop; otherwise the resolution process is carried on iteratively. The actual learning is performed by `add_clause_to_database()`. The generated clause is redundant and can be added back to the clause database to help prune search in the future.

To use this procedure in a QBF solver, some complications arise. In the SAT case, when the function `resolve()` is called, one of the input clauses is a conflicting clause (all literals evaluate to 0), and another is the antecedent of a literal in that clause (all but one literals evaluate to 0). Therefore, the distance between these two clauses is 1 and the resolution process can work on these two clauses. The resolvent of these two clauses has all literals evaluating to 0 and is still a conflicting clause. Therefore, the resolution process can go on iteratively. This is not true for QBF solvers because QBF uses different deduction rules. For example, consider two clauses in a QBF formula (there are other clauses in the QBF, but only two of them are shown):

$$(a_{(1)} + b_{(3)} + x_{(4)} + c_{(5)})(a_{(1)} + b'_{(3)} + x'_{(4)} + d_{(5)})$$

a, b, c, and d are existential variables, x is a universal variable. The numbers in the subscripts are the quantification levels for the corresponding variables. Suppose initially, c and d are both implied to 0 at decision level 5, and all the other literals are free. At decision level 6, suppose the solver makes a branch $a = 0$. By the implication

rule (Rule 1), the first clause is unit because x has a higher quantification level than b, and c is already 0. Therefore, b is implied to be 1. This implication results in the second clause to become conflicting (because a, b', and d are all 0). The function `analyze_conflict()` calls `resolve()`. The result of the resolution of the second clause with the antecedent of b (i.e. the first clause) gives:

$$(a_{(1)} + x_{(4)} + x'_{(4)} + c_{(5)} + d_{(5)}).$$

This clause is a tautology clause because it has both x and x' as literals. This is not a surprise because the two original clauses have a distance greater than 1. It seems that because of this, the resolution process cannot continue. On the contrary, this section shows that such a tautology clause generated from resolution during conflict analysis process can be treated as a regular non-tautology clause and the resolution process can continue as in the SAT case.

Due to these complications, it is not obvious that the procedure depicted in Figure 5.6 can be used for QBF solving. The next subsection will show that the tautology clauses generated by the procedure actually obey the same rules for conflict and implication, and can be regarded as regular non-tautology clauses for these purposes. Therefore, the procedure is in fact applicable in a QBF solver.

5.3.3 Long Distance Resolution

This subsection proves that it is possible to regard a resolvent clause between two clauses with distances greater than 1 as a regular (i.e., non-tautology) clause, and this clause obeys the same unit implication rule and conflict rule (described by Rules 1 and 2) as regular non-tautological clauses. In the example shown in the previous subsection, because a has a lower quantification level than x; and c and d are assigned 0 at decision level 5, the last clause is a unit clause at decision level 5. Therefore, a should be implied at decision level 5 with value 1. Note that this does not follow

immediately from Rule 1, as this clause is a tautology clause. In particular the proof for the implication rule for non-tautology clauses [21] does not apply here. Instead, this needs to be proved as a totally new result.

The proof strategy is outlined here first, and a more rigorous proof follows. Clauses in a QBF solver have two functions. The first function is to provide non-conflicting implications, and thus lead the search to a new space, the second is to show conflicts, and thus declare that a certain search space has no solution. Therefore, to prove that clauses (regardless of tautology or not) generated from resolution can be regarded as regular clauses, it is necessary to prove that they can be used to generate implications and conflicts by the same implication rule and conflict rule. The main effort of the proof is to show that if both of the two input clauses to the function `resolve()` in Figure 5.6 obey these two rules, then the output clause also obey these two rules even if the clause is a tautology clause. Initially, the entire clause database consists of non-tautology clauses, and they obey both rules. Therefore, by induction, any clauses generated from the function `analyze_conflict()` during the solving process described in Figure 5.5 also obey these two rules. Therefore, they can be regarded as regular non-tautological clauses for implication and conflict generation purposes in future search.

To prove the results, some terms need to be defined and some assumptions need to be formalized.

Definition 2: ***Distance between two sets of literals***

Similar to the notion of distance between cubes or clauses, the distance between two sets of literals ω_1 *and* ω_2 *is the number of literals* l *such that* $l \in \omega_1$ *and* $l' \in \omega_2$. $D(\omega_1, \omega_2)$ *will be used to denote the distance between* ω_1 *and* ω_2.

Moreover, the rules that will be used need to be described. These two rules are the same as the Rules 1 and 2 except that they are relaxed and do not require that

the clauses be non-tautology.

Proposition 3: ***Implication Rule for Clauses***

For a QBF formula F in the form of 5.33, if at a certain node in the search tree for a DLL QBF solver following the process depicted in Figures 5.5 and 5.6, a clause C has a literal e such that:

1. $e \in E(C), V(e) = X$. *For any* $e_1 \in E(C), e_1 \neq e : V(e_1) = 0$;

2. $\forall u \in U(C), V(u) \neq 1$. *If* $V(u) = X$, *then* $L(u) > L(e)$

Then the value of the current node is the same as the QBF formula resulting from replacing the literal e of the formula the current node represents with value 1.

Proposition 4: ***Conflicting Rule for Clauses***

For a QBF F in the form of 5.33, if at a certain node in the search tree for a DLL QBF solver following the process depicted in Figures 5.5 and 5.6, there exists a non-tautology clause C, such that $\forall e \in E(C), V(e) = 0$, and $\forall u \in U(C), V(u) \neq 1$, then the current node evaluates to 0.

Because of the relaxation of the non-tautological condition, these two rules only hold if the search and learning follow the pseudo-codes of Figure 5.5 and Figure 5.6. That is why they are explicitly mentioned in the Propositions.

This following lemma is used to help prove these two propositions.

Lemma 4: *If clause C_1 and C_2 both follow the Conflicting Rule (Proposition 4) and Implication Rule (Proposition 3), and they satisfy:*

1. $D(E(C_1), E(C_2)) = 1$. *Let the distance 1 existential literal be e, i.e.* $e \in E(C_1), e' \in E(C_2)$.

2. *For any universal literal* u*, if* $u \in U(C_i)$ *and* $u' \in U(C_j), i, j \in \{1, 2\}$ *then* $L(u) > L(e)$.

Then the resolvent clause C *with* $E(C) = (E(C_1) \setminus e) \cup (E(C_2) \setminus e')$ *and* $U(C) = U(C_1) \cup U(C_2)$ *also obeys these two rules.*

Proof: To simplify the notation, define $E_e(C_1) = E(C_1) \setminus e, E_{e'}(C2) = E(C_2) \setminus e'$. Now each rule is proved separately.

a) Clause C obeys the Conflicting Rule.

Because of condition 1 in Lemma 4, $D(E_e(C_1), E_{e'}(C_2)) = 0$. Therefore, there may be some variable assignments that satisfy the conditions of the Conflicting Rule, i.e., the assignment can make $\forall e_1 \in E(C), V(e_1) = 0, \forall u_1 \in U(C), V(u_1) \neq 1$. Now we need to prove that if that is the case, the QBF is unsatisfiable at this branch (i.e., the QBF corresponding to current node in the tree evaluates to 0). There are three cases:

1. If $V(e) = 0$, C_1 satisfies the condition of the Conflicting Rule. So the current node evaluates to 0.

2. If $V(e) = 1$, C_2 satisfies the condition of the Conflicting Rule. So the current node evaluates to 0.

3. If $V(e) = X$, it is possible to demonstrate an assignment of universal variables that makes the propositional formula evaluate to 0, regardless of how the existential variables are assigned. For any $u \in U(C), L(u) < L(e)$, assign the variable corresponding to u such that $V(u) = 0$. This can be done because condition 2 of Lemma 4 guarantees that if $L(u) < L(e)$ and $u \in U(C)$, then $u' \notin U(C)$. Then, both clauses C_1 and C_2 satisfy the condition of the Implication Rule. Clause C_1 requires $V(e) = 1$, while clause C_2 requires $V(e') = 1$. This leads to conflict, and the current node evaluates to 0.

Therefore, the Conflicting Rule holds.

b) Clause C obeys the Implication Rule.

Suppose there exists $e \in E(C)$, for any $e_1 \in E(C)$ and $e_1 \neq e$, $V(e_1) = 0$. Moreover, for any $u \in U(C)$, $V(u) \neq 1$; if $V(u) = X$, then $L(u) > L(e)$. Now we need to prove that such a situation implies that current node has the same value that results by assigning $V(e) = 1$.

Because e has a smaller quantification level than that of any of the free universal literals appearing in C, it must appear at a higher position than any of them in the semantic tree, i.e. it must be assigned before any of them. Therefore, if the assignment makes $V(e) = 0$, then the clause C satisfies the condition of the Conflict Rule. Thus, by the first part of the proof, it is a conflict. Therefore, F can be satisfied in the branch only if $V(e) = 1$. Therefore, the current node has the same value that results by assigning $V(e) = 1$.

Therefore, the Implication Rule also holds. ■

By using Lemma 4, it is possible to prove the two rules in Propositions 3 and 4 as follows:

Proof: The proof is by induction on the resolution depth of the resulting clause. Resolution depth is defined recursively. The resolution depth of the clause generated from the `resolve()` function is the maximum resolution depth of the two input clauses plus 1. Initially, all the original clauses have resolution depth 0.

Induction Basis: Since none of the input clauses are tautological clauses, if a clause has resolution depth 0, it satisfies both rules. This result is the application of Rules 1 and 2 shown in Section 5.2.2 [21].

Induction Hypothesis: The rules holds for resolution depth smaller than or equal to n.

Induction Step: Clause C with resolution depth of $n + 1$ is obtained by calling

`resolve()` on two clauses C_1 and C_2. Both C_1 and C_2 have resolution depth less than or equal to n (one of them must be n). Therefore, by the induction hypothesis, both C_1 and C_2 follow the Implication Rule and the Conflicting Rule in Propositions 3 and 4 respectively. We now prove that these two clauses satisfy both conditions 1 and 2 of Lemma 4, and thus the rules holds for the resulting clause.

Suppose C_2 is the antecedent of the resolution variable var, and the corresponding literal for var is a. Then obviously, $V(a) = 1, \forall e \in E(C_2), e \neq a, V(e) = 0$. C_1 is obtained from a conflicting clause by a sequence of resolution steps. All the other clauses involved in these resolution steps are unit clauses, and the unit literals are removed during the resolution process. Therefore, $\forall e \in E(C_1), V(e) = 0$. Thus $D(E(C_1), E(C_2)) = 1$. Therefore, condition 1 of Lemma 4 is satisfied.

Suppose there is a literal $u \in U(C_2)$. If $u' \in U(C_2)$, then $L(u) > L(a)$ because otherwise a cannot be implied by this clause. If $u' \in U(C_1)$, then $L(u) > L(a)$ because otherwise, for a to be implied in C_2, $V(u) = 0$. But then $V(u') = 1$, therefore the clause that has u' in it is neither a unit clause nor a conflicting clause, so this literal should not appear in C_1. Thus Condition 2 of Lemma 4 is satisfied.

Because both Conditions 1 and 2 of Lemma 4 are met, the resulting clause C obeys both rules. By induction, the rules hold. ■

Notice that the proof requires only that $D(E(C_1), E(C_2)) = 1$, i.e. the existential part of the two clauses has distance 1. There is no requirement on the distance between the universal literals of the two clauses. Therefore, the distance between these two clauses may be larger than 1. Such a resolution between two clauses with distance larger than 1 will be called *long-distance resolution*. The proof shows that the solver can just regard the results from the resolution process as regular non-tautology clauses for the purposes of implication and conflict, even though some of these resolvent clauses may contain universal literals in both phases.

Büning *et. al.* propose a concept called *Q-resolution* [18]. Q-resolution is similar to regular resolution except that it omits those universal variables in the resulting clause that have higher quantification level than any existential variable in the clause. For example, assuming a_2, b_4 and c_6 are existential variables, x_5 and y_3 are universal variables, and the numbers in the subscripts are the quantification levels for the corresponding variables, then the resolvent of clauses $(a_2+b_4+x_5+c_6)$ and $(y_3+b_4+c'_6)$ is $(a_2+y_3+b_4+x_5)$ while the result of Q-resolution for these two clauses is $(a_2+y_3+b_4)$. Therefore, in Q-resolution, the resulting clause may contain fewer variables than an ordinary resolvent from the same two clauses. Q-resolution is valid in QBF [18], i.e. the result from a Q-resolution is redundant and can be added to the original clause database. The proof for the two propositions is valid even if "Q-resolution" is used instead of regular resolution. Intuitively, the trailing universal variables in a clause are not important for determining the satisfiability of the QBF because in the "game" point of view, they always try to falsify the formula; and none of the existential variables can prevent them because existential variables in the clause must all be assigned before the trailing universal variables. Therefore, if the formula is satisfiable, the existential variables must already satisfy the clause, before the assignments of the trailing universal variables, so the trailing universal variables do not matter anymore.

5.3.4 Stopping Criteria for Conflict Analysis

The resolution process of conflict analysis is run iteratively until some stopping criteria are met. The stopping criteria make sure that the resulting clause from the resolution is a clause that can actually resolve the current conflict, and bring the search to a new space. In the SAT case, the stopping criterion is that the resulting clause be *asserting*, i.e. the clause will become a unit clause after backtracking. In the QBF case, the stopping criteria are a little different. The resolution process for conflict analysis should stop if the generated clause satisfies the following conditions:

1. Among all its existential variables, one and only one of them has the highest decision level. Suppose this variable is v.

2. v is at a decision level with an existential variable as the decision variable.

3. All universal literals in the clause with quantification levels smaller than v's are assigned 0 at decision levels smaller than the decision level of v.

The first and third conditions make sure that this clause is indeed an asserting clause (i.e. unit after backtracking). The second condition makes sure that the solver will undo at least one existential decision during the backtrack because undoing universal decisions only can never completely resolve a conflict. An exception to this stopping criterion is that if all existential literals in the resulting clause are at decision level 0, or if there are no existential literals in the resulting clause, then the solver should stop and immediately state that the problem is unsatisfiable.

Now that the stopping criteria for the resolution is described, the algorithm for conflict-driven learning and non-chronological backtracking for a QBF solver working on a CNF database is complete. This algorithm is essentially an adaptation of the conflict-driven learning and non-chronological backtracking algorithm used in SAT solvers. In the next section, satisfaction-driven learning and non-chronological backtracking will be described.

5.4 Satisfaction-Driven Learning

This section examines learning and non-chronological backtracking for satisfying leaves in a DLL-based QBF solver[2]. The last section examined conflict-driven learning and non-chronological backtracking in a QBF solver. In this section, similar ideas

[2]Part of the work described in this section was first reported in CP 2002 [140].

will be applied on satisfying leaves. However, to accomplish this, new data structures and new deduction rules need to be introduced.

This section is organized as follows. Section 5.4.1 describes an enhanced data structure to accommodate satisfaction-driven learning and non-chronological backtracking. A CNF database is augmented with *cubes* and the rationale behind such augmentation is discussed. The deduction rules that can be used with the new data structure are provided in Section 5.4.2. Section 5.4.3 discusses how to generate redundant cubes from a CNF formula. In Section 5.4.4, satisfaction-driven learning and non-chronological backtracking is discussed.

5.4.1 Augmented Conjunctive Normal Form (ACNF)

As mentioned before, traditionally QBF solvers require that the propositional part of the QBF be in Conjunctive Normal Form (CNF) as shown in Equation 5.33. However, CNF is not the only way to represent a propositional formula. As seen in Chapter 2, a propositional formula can be in Disjunctive Normal Form (DNF) as well. Suppose a propositional formula φ has its CNF and DNF representation as:

$$\varphi = C_1 C_2 \ldots C_k = S_1 + S_2 + \ldots + S_l$$

Then:

$$\begin{aligned}
\varphi &= C_1 C_2 \ldots C_k && (5.34)\\
&= S_1 + S_2 \ldots + S_l && (5.35)\\
&= C_1 C_2 \ldots C_k + S_1 + S_2 \ldots + S_l && (5.36)\\
&= C_1 C_2 \ldots C_k + \Sigma AnySubSet\{S_1, S_2 \ldots S_l\} && (5.37)\\
&= \Pi AnySubSet\{C_1, C_2 \ldots C_k\}(S_1 + S_2 \ldots + S_l) && (5.38)
\end{aligned}$$

Here, $\Sigma\omega$ is used to denote disjunction of elements in set ω, and $\Pi\omega$ is used to denote conjunction of the elements in the set ω. $AnySubSet(\omega)$ is used to denote any

set ν s.t. $\nu \subseteq \omega$. A propositional formula in form 5.37 is called a propositional formula in *Augmented Conjunctive Normal Form (ACNF)* and a formula in form 5.38 is called a formula in *Augmented Disjunctive Normal Form (ADNF)*. If the propositional part of a QBF is in Augmented Conjunctive Normal Form, the QBF is said to be in Augmented Conjunctive Normal Form. Because the ACNF will be used extensively in future discussion, it is defined here.

Definition 3: *Augmented CNF*

A Propositional formula φ is said to be in Augmented CNF (ACNF) *if*

$$\varphi = C_1 C_2 \dots C_m + S_1 + S_2 + \dots + S_k$$

where C_i's are clauses and S_i's are cubes. Moreover, each S_i is contained in the term $C_1 C_2 \dots C_m$, i.e.,

$$\forall j \in \{1, 2 \dots k\}, S_j \Rightarrow C_1 C_2 \dots C_m$$

A quantified Boolean formula in the form of Equation 5.32 is said to be in Augmented Conjunctive Normal Form (ACNF) *if the propositional formula φ is in ACNF. The conjunction of all the clauses $C_1 C_2 \dots C_m$ in the ACNF is called the* clause term *of the ACNF.*

Notice that from the definition of ACNF, each of the cubes is contained in the clause term. Therefore, the cubes are actually redundant in ACNF. Deleting any or all of the cubes will not change the Boolean function represented by the propositional formula φ or the quantified Boolean formula F. Notice here the clause term contains all the information about φ, but the cubes do not.

In previous works (e.g. [21, 107, 42]), the QBF solvers operate on a clause database that corresponds to a propositional formula in CNF. All the deduction rules are valid under the implicit condition that the QBF is in CNF. This section discusses a QBF solver that operates on an ACNF database, with both clauses and cubes presented in

the database. Because CNF is a special case of ACNF, any rules that are applicable to ACNF are also valid for CNF. By using ACNF, it is possible to introduce new rules for the solver to prune the search space further and the solver can incorporate satisfaction-driven learning and non-chronological backtracking which is symmetric to conflict-directed learning and non-chronological backtracking.

The following discussion provides the intuition behind the augmentation of the CNF database in a QBF solver.

Traditionally, for SAT, the DPLL algorithm requires that the formula be in CNF. The reasons for that are the two important rules that are direct results for formulas in CNF: the *unit literal rule* and the *conflicting rule*. The unit literal rule states that if an unsatisfied clause has only one free literal, then it must be assigned to value 1. The conflicting rule states that if a clause has all literals evaluating to 0, then the current branch is not satisfiable. The function of these two rules is to direct the solver away from searching space that will not affect the outcome of the search because the result of the search in that space is *obvious*. In the SAT case, the obvious outcome is that there is no satisfiable assignment in that space. For example, if there exists a conflicting clause, then the entire Boolean space corresponding to the current assignment will not satisfy the formula, so the solver should backtrack immediately. If there exists a unit clause, then assigning the unit literal with value 0 will lead to a search space with no solution, so the solver should assign it 1. In SAT, the solver is interested in finding only one satisfying assignment. Therefore, it needs to prune only the space with no solution. The implications by unit clauses are called *conflict directed* implications because the purpose of the implication is to avoid conflicts (i.e., the no-solution space).

A QBF solver is different from a SAT solver because it usually cannot stop when a single satisfying branch for the propositional part of the formula is found. In fact, it needs to search multiple combinations of the assignments to universal variables to

determine satisfiability. Therefore, the solver is not only interested in pruning the space that obviously has no satisfying assignment for the propositional part, it is also interested in pruning the space that obviously *has* a solution for the formula. Previous DLL based QBF solvers are mostly based on the works of Cadoli *et al.* [21], which in turn is based on SAT procedures and require the database in CNF. Even though the implication rules and the conflicting rules for QBFs are a little different from SAT, these rules are still conflict-directed, i.e., they take the search away from an obviously no-satisfying-assignment space. There is no mechanism in these algorithms to take the search away from an obviously has-solution space.

It is to cope with this obvious asymmetry that the Augmented Conjunctive Normal Form (ACNF) is introduced. In ACNF, cubes are **or**-ed with the clause term. Whenever a cube is satisfied, the entire propositional formula evaluates to 1. Similar to unit clauses, it is possible to have the notion of *unit cubes*. A unit cube will generate an implication, but the purpose of the implication is to take the search away from a space that obviously has solutions. Similar to conflicting clauses, it is possible to have a notion of *satisfying cubes*. Whenever a satisfying cube is encountered, the solver can declare that the branch is satisfiable and backtrack immediately. Implications and satisfactions on cubes have rules similar to implications and conflicts on clauses. The rules for the cubes will be discussed in subsequent sections.

Most frequently the QBF instance is presented to the solver in CNF form. Therefore, initially there is no cube in the database. The solver can generate cubes during the solving process. ACNF requires that the generated cubes be contained in the clause term. Because of this requirement, the satisfiability of the QBF will not be altered by these cubes. This is similar to conflict-driven learning in SAT. In SAT, even though the solver adds learned clauses to the original CNF, the satisfiability of the original CNF will not change. However, by adding these redundant clauses, the solver may be able to prune the search space in the future. Similarly, here in QBF,

even though the cubes are redundant with respect to the original clause term, by keeping them in the database, the solver may be able to use them to prune the search space in the future. The next section will discuss the rules for cubes, assuming the database is already in ACNF. The way to generate cubes during the solving process, staring from a CNF form, will be discussed in Section 5.4.3.

5.4.2 Implication Rules for the Cubes

This section shows the rules used in the `deduce()` function shown in Figure 5.5. These rules are valid for QBF in ACNF forms. Therefore, they can be directly applied to the database the solver is working on.

The implication rule for non-tautology clauses shown in Rule 1 and the conflicting rule shown in Rule 2 are valid. When conflict-driven learning is enabled, these two rules, stripped of the requirement of the clauses being non-tautology, are still valid (from Propositions 3 and 4). The reason is that the cubes are redundant with respect to the clause term. Therefore, adding them does not invalidate rules that are valid in a CNF database. There are two corresponding rules for the cubes as well. Recall that $S, S_1, S_2 \ldots$ are used to represent cubes. A cube is *empty* if it contains both a literal and its complement, otherwise, it is non-empty. The following rules apply to non-empty cubes.

Rule 5: ***Unit Implication Rule for Non-Empty Cubes.***

For a QBF formula F in the form of 5.37, if at a certain node in the search tree, a non-empty cube S has a literal u such that:

1. $u \in U(S), V(u) = X$. *For any* $u_1 \in U(S), u_1 \neq u : V(u_1) = 1$;

2. $\forall e \in E(S), V(e) \neq 0$. *If* $V(e) = X$, *then* $L(e) > L(u)$

Then the value of the current node is the same as the QBF formula resulting from replacing all the occurrences of literal u with value 0 in the formula that the current node represents.

The implication rule for cubes tries to avoid searching the other branch of valuation for literal u, when that would not affect the outcome of the current node. A cube satisfying the two conditions in Rule 5 is called a *unit cube*. When a literal is forced to be assigned with a value because it appears in a unit cube, the literal is said to be *implied*, and the unit cube is the *antecedent* of the literal. Note that cubes only imply universal variables, and the antecedent of a universal variable is always a cube.

Rule 6: ***Satisfying Rule for Non-Empty Cubes.***

For a QBF F in the form of 5.37, if in a certain node in the search tree, there exists a non-empty cube S, such that $\forall u \in U(S), V(u) = 1$, and $\forall e \in E(S), V(e) \neq 0$, then the current node will evaluate to 1.

When a non-empty cube satisfies the condition of Rule 6, the cube is said to be *satisfying*. If there exists a satisfying cube, then it is not necessary to search the rest of the subtree. The solver can label the current node to be 1 and backtrack immediately.

Rules 5 and 6 are the rules for implication and satisfaction for non-empty cubes. They are exactly the duals of Rules 1 and 2 respectively, which are for clauses. When the QBF solver operates on an ACNF database, these two rules are used in the function `deduction()` and `is_satisfied()` respectively to prune search space. The pseudo-code for these functions are listed in Figure 5.7.

5.4.3 Generating Satisfaction-Induced Cubes

This section discusses how to generate cubes that are contained by the clause term in an ACNF database. When the QBF problem is given to the solver, it usually is

```
deduce() {
    while(!is_conflicting() && !is_satisfying()) {
        if (exist_unit_clause())
            assign_unit_literal_in_clause_to_be_1();
        else if (exist_unit_cube())
            assign_unit_literal_in_cube_to_be_0();
        else if (exist_pure_literal())
            assign_pure_literal_with_proper_value();
        else
            return UNDETERMINED;
    }
    if (is_satisfying())
        return SAT;
    return CONFLICT;
}

is_conflicting() {
    if (exist_conflicting_clause())
        return true;
    return false;
}

is_satisfied() {
    if (exist_satisfying_cube())
        return true;
    else
        if (all_clause_has_at_least_one_value_1_literal())
            return true;
    return false;
}
```

Figure 5.7: The function for deduction on an ACNF database

in CNF and does not have any cubes. To generate cubes that are contained by the clause term, one obvious way is to expand the clause term using the distribution law. Unfortunately, this is not practically feasible because the number of cubes generated is intractable.

This section discusses another way to generate cubes. The main idea is that whenever the search procedure finds that all the clauses are satisfied (i.e. for each clause, at least one literal evaluates to 1), it is always possible to find a set of value 1 literals such that for any clause, at least one of the literals in the set appears in it. Such a set is called a *cover set* of the satisfying assignment. The conjunction of the literals in the cover set is a cube, and this cube is guaranteed to be contained by the clause term.

For example, consider the ACNF:

$$(a+b+x)(c+y')(a+b'+y')(a+x'+y')+xy'$$

The variable assignment $\{a=1, b=0, c=X, x=0, y=0\}$ is a satisfying assignment. The set of literals $\{a, y'\}$ is a cover set. Therefore, cube ay' is a cube that is contained in the clause term, and can be added to the ACNF. The resulting formula will be:

$$(a+b+x)(c+y')(a+b'+y')(a+x'+y')+ay'+xy'$$

It is easy to verify that the cube ay' is indeed contained by the clause term. A cube generated from a cover set of a satisfying assignment will be called a *satisfaction-induced cube*.

For a satisfying assignment, the cover set is not unique. Therefore, it is possible to generate many satisfaction-induced cubes. Which and how many of these cubes should be added to the database is to be determined by heuristics. Different heuristics, while not affecting the correctness of the algorithm, may greatly affect the

efficiency. Evaluating different heuristics for generating satisfaction-induced cubes is an interesting topic for future work. Here we will simply assume that some heuristics (for example, greedily find a small set) can be used to choose a single covering set for each satisfying assignment.

5.4.4 Satisfaction Analysis in a QBF Solver

When a satisfying leaf is encountered, it is possible to perform satisfaction-driven learning similar to conflict-driven learning. Conflict-driven learning adds (redundant) clauses to the ACNF database. Similarly, satisfaction-driven learning adds (redundant) cubes to the ACNF database. The procedure for satisfaction-driven learning, which is shown in Figure 5.8, is very similar to the procedure for conflict-driven learning. The only difference is that when a satisfiable leaf is encountered, there are two scenarios, while in the conflicting case there is only one. The first scenario in the satisfying leaf case is that there exists a satisfying cube in the ACNF; this is similar to the conflicting case, where there exists a conflicting clause. The second scenario, which is unique to the satisfying case, is that all the clauses in the ACNF are satisfied (i.e. every clause has at least one literal evaluate to 1) but no satisfying cube exists. In Figure 5.8, if function `find_sat_cube()` returns NULL, then the second case is encountered. In that case, the solver has to construct a satisfaction-induced cube from the current variable assignment. The learned cube is generated in a manner similar to the learned clauses by replacing resolution with consensus.

Similar to the conflict analysis case, the stopping criteria for the satisfaction analysis are:

1. Among all its universal variables, one and only one of them has the highest decision level. Suppose this variable is v.

2. v is at a decision level with a universal variable as the decision variable.

```
analyze_satisfiability(){
    if (current_decision_level()==0)
        return -1;
    s = find_satisfying_cube();
    if (s == NULL)
        s = construct_satisfaction_induced_cube();
    while (!stop_criterion_met(s)) {
        lit = choose_literal(s);
        var = variable_of_literal( lit );
        ante = antecedent( var );
        s = consensus(s, ante, var);
    }
    add_cube_to_database(s);
    back_dl = cube_asserting_level(s);
    return back_dl;
}
```

Figure 5.8: Satisfaction analysis in a QBF solver

3. All existential literals in the cube with quantification levels smaller than v's are assigned 1 at decision levels smaller than the decision level of v.

The first and third conditions make sure that this cube is an asserting cube. The second condition makes sure that the solver will undo at least one *universal* decision during the backtrack. An exception to this stopping criterion is that if all universal literals in the resulting cube are at decision level 0, or if there is no universal literal in the resulting cube, then the solver should stop and immediately state that the problem is satisfiable.

Similar to the conflict analysis case, it is also possible to prove that the cubes generated from the satisfaction-driven learning procedure are valid and obey the same deduction rules, even if some of the cubes may be empty. Similar to "Q-resolution" discussed in Section 5.3.3, it is also possible to introduce the notion of "Q-consensus". Q-consensus is similar to regular consensus except that it omits those

existential variables in the resulting cube that have higher quantification level than any universal variable in the cube. Using "Q-consensus" instead of regular consensus in the procedure depicted in Figure 5.8, the result still holds. It is also easy to prove that the pure literal rules corresponding to Rules 3 and 4 are still valid for an ACNF database, by regarding the solution process as a "game".

Theorem 2: *The algorithm depicted in Figure 5.5, when working on an ACNF database, is correct. Moreover, clauses and cubes generated by algorithm depicted in Figures 5.6 and 5.8 obey the Rules 1,2,3,4,5,6 even if the clauses are tautology clauses or the cubes are empty cubes.*

From the pseudo-code of `analyze_conflict()` and `analyze_satisfiability()`, the DLL procedure will have an almost symmetric view on satisfying and conflicting leaves in the search tree. Whenever a conflicting leaf is encountered, a clause will be learned to prune the space that obviously has no solution. Whenever a satisfying leaf is encountered, a cube will be learned to prune the space that obviously has solutions. The only asymmetry that still exists is that in the database, the cubes are contained by the clause term, but not vice-versa. Therefore, the cubes only contain partial information about the propositional formula. Because of this, the solver may need to generate satisfaction-induced cubes when satisfying assignments are found, but no existing cube is satisfied.

The propositional formulas need not be limited to ACNF for QBF. If the original QBF problem is given in DNF form, it may be augmented to ADNF (note that problem in DNF is trivial for SAT). In that case, the solver may need to construct conflict-induced clauses. All the other aspects of the solver will still be the same.

5.5 Experimental Results

The algorithm described in this chapter is implemented in a QBF solver called Quaffle, which loosely stands for Quantified Formula Evaluator with Learning. Quaffle is implemented in C++. This section compares the performance of four versions of Quaffle. The first version implements the naïve version of the DLL procedure for QBF solving, i.e. chronological backtracking with no learning. The second version includes non-chronological backtracking driven by conflicts, but no learning. This corresponds to implementing `analyze_conflict()` with the function `add_clause_to_database()` disabled. This will be referred to as the BJ (Back Jumping) version. In this version, when a satisfying leaf is encountered, the solver will backtrack naïvely. The third version is the same as the second except that learning is turned on. It will be called the CDL (Conflict Driven Learning) version. The last version implements both conflict-driven, as well as satisfaction-driven learning and non-chronological backtracking. It will be referred to as the Full version.

The benchmark suite used for evaluation is obtained from J. Rintanen [110]. The benchmarks consist of some random QBF instances as well as some instances generated from real applications from the AI domain. To compare with the state-of-the-art, some of the existing QBF solvers are also evaluated. Statistics for two of the most efficient solvers for the test suite used are listed in the tables. QuBE-BJ is the backjumping [41] version of QuBE [42], while QuBE-BT is the version with simple backtracking scheme. Decide [108] is the solver developed by J. Rintanen. The experiments are conducted on a Dell PIII 933 machine with 1G memory. The timeout limit is 1800 seconds for each instance. The instances for which all the solvers aborted due to timeout are omitted. The results are shown in Tables 5.1 to 5.3.

Testcase	#Var	#Cls	T/F	Quaffle				Qube		Decide
				Naïve	BJ	CDL	Full	BT	BJ	
BLOCKS3i.4.4	288	2928	F	-	0.07	0.07	0.09	-	-	0.05
BLOCKS3i.5.3	286	2892	F	-	128.42	29.03	103.73	-	-	31.89
BLOCKS3i.5.4	328	3852	T	-	30.75	2.88	146.54	-	-	6.59
BLOCKS3ii.4.3	247	2533	F	-	0.07	0.05	0.04	-	5.02	0.03
BLOCKS3ii.5.2	282	2707	F	-	0.82	0.13	0.48	-	82.19	0.07
BLOCKS3ii.5.3	304	3402	T	-	2	0.33	0.48	-	-	1.5
BLOCKS3iii.4	202	1433	F	-	0.05	0.03	0.03	-	1.62	0.01
BLOCKS3iii.5	256	1835	T	-	0.72	0.27	0.23	-	-	0.26
BLOCKS4i.6.4	779	15872	F	-	-	249.09	110.2	-	-	5.35
BLOCKS4ii.6.3	838	15061	F	-	-	367.54	591.95	-	-	4.53
BLOCKS4ii.7.2	783	15219	F	-	-	-	-	-	-	9.15
BLOCKS4ii.7.3	863	18768	T	-	-	-	-	-	-	731.24
BLOCKS4iii.6	727	9661	F	-	-	-	-	-	-	2.92
BLOCKS4iii.7	855	11303	T	-	-	-	-	-	-	414.76
CHAIN12v.13	925	4582	T	-	0.32	0.31	7.11	0.36	0.34	0.41
CHAIN13v.14	1080	5458	T	-	0.69	0.66	19.09	0.84	0.8	0.79
CHAIN14v.15	1247	6424	T	-	1.49	1.45	51.09	2.45	2.27	1.6
CHAIN15v.16	1426	7483	T	-	3.25	3.15	142.21	4.57	5.17	3.25
CHAIN16v.17	1617	8638	T	-	7.03	6.82	472.38	15.53	13.38	6.7
CHAIN17v.18	1820	9892	T	-	15.14	14.85	1794.35	34.89	43.44	14.48
CHAIN18v.19	2035	11248	T	-	33.02	32.4	-	97.73	100.06	31.21
CHAIN19v.20	2262	12709	T	-	75.09	71.41	-	234.65	249.04	61.08
CHAIN20v.21	2501	14278	T	-	166.98	154.86	-	527.92	650.44	130.71
CHAIN21v.22	2752	15958	T	-	362.24	343.62	-	1271	1462.59	272.58

Table 5.1: Runtime comparison of different QBF solvers Pg. 1 (unit: seconds)

Testcase	#Var	#Cls	T/F	Quaffle				Qube		Decide
				Naïve	BJ	CDL	Full	BT	BJ	
CHAIN22v.23	3015	17752	T	-	846.08	747.3	-	-	-	569.65
CHAIN23v.24	3290	19663	T	-	1790.14	1710.06	-	-	-	1200.91
impl02	10	18	T	0.00	0.00	0.00	0.00	0.00	0.00	0.00
impl04	18	34	T	0.00	0.00	0.00	0.00	0.00	0.00	0.01
impl06	26	50	T	0.01	0.01	0.00	0.00	0.00	0.01	0.04
impl08	34	66	T	0.07	0.04	0.01	0.01	0.01	0.00	0.29
impl10	42	82	T	0.48	0.33	0.02	0.01	**	**	2.29
impl12	50	98	T	3.57	2.34	0.06	0.01	**	**	17.38
impl14	58	114	T	26.41	18.19	0.24	0.02	**	**	130.64
impl16	66	130	T	195.52	135.22	0.97	0.02	**	**	974.53
impl18	74	146	T	1445.76	985.57	3.88	0.02	**	**	-
impl20	82	162	T	-	-	15.51	0.02	-	-	-
lognBWLARGEA0	828	1685	F	0.00	0.00	0.00	0.00	0.00	0.00	0.01
lognBWLARGEA1	1099	62820	F	-	67.47	2.21	2.14	-	7.28	0.47
lognBWLARGEA2	1370	65592	T	-	-	125.85	193.88	-	-	28.28
lognBWLARGEB0	1474	3141	F	0.01	0.01	0.00	0.00	0.00	0.00	0.03
lognBWLARGEB1	1871	178750	F	-	342.55	8.26	8.18	-	37.89	0.61
lognBWLARGEB2	2268	183601	F	-	-	763.37	750.92	-	-	1.88
R3CNF_150_3_2.50_0.T	150	375	T	-	3.15	1.22	0.02	0.00	0.00	0.03
R3CNF_150_3_2.50_1.F	150	375	F	-	2.52	0.02	0.05	0.03	0.01	0.47
R3CNF_150_3_2.50_2.T	150	375	T	738.7	1.11	0.81	0.01	0.00	0.00	0.05
R3CNF_150_3_2.50_3.T	150	375	T	521.98	1.65	1.06	0.00	0.11	0.00	0.06
R3CNF_150_3_2.50_4.T	150	375	T	-	2.74	1.43	0.09	0.06	0.01	0.1

Table 5.2: Runtime comparison of different QBF solvers Pg. 2 (unit: seconds)

Testcase	#Var	#Cls	T/F	Quaffle				Qube		Decide
				Naïve	BJ	CDL	Full	BT	BJ	
R3CNF_150_3_2.50_5.T	150	375	T	71.41	21.42	0.95	0.06	0.01	0.00	0.07
R3CNF_150_3_2.50_6.F	150	375	F	-	11.99	1.51	0.37	0.07	0.03	20.52
R3CNF_150_3_2.50_7.F	150	375	F	-	7.88	0.6	0.07	0.05	0.02	1.91
R3CNF_150_3_2.50_8.F	150	375	F	-	1.33	0.29	0.05	0.22	0.06	0.32
R3CNF_150_3_2.50_9.T	150	375	T	40.99	1.03	0.87	0.02	0.05	0.01	0.25
R3CNF_150_7_2.60_0.F	150	390	F	479.68	2.19	0.14	0.11	0.44	0.02	0.01
R3CNF_150_7_2.60_1.T	150	390	T	0.55	0.52	0.23	0.02	0.07	0.01	0.06
R3CNF_150_7_2.60_2.T	150	390	T	423.71	0.65	1.27	0.02	0.00	0.00	0.05
R3CNF_150_7_2.60_3.T	150	390	T	36.83	0.51	0.34	0.02	0.75	0.02	0.15
R3CNF_150_7_2.60_4.T	150	390	T	-	51.64	13.3	0.17	0.76	0.01	0.04
R3CNF_150_7_2.60_5.F	150	390	F	-	8.32	1.3	0.11	0.64	0.01	27.12
R3CNF_150_7_2.60_6.T	150	390	T	150.08	1.79	0.51	0.03	0.19	0.03	0.83
R3CNF_150_7_2.60_7.T	150	390	T	-	11.9	2.21	0.33	0.73	0.06	0.18
R3CNF_150_7_2.60_8.F	150	390	F	1.85	0.00	0.00	0.00	0.02	0.01	0.08
R3CNF_150_7_2.60_9.T	150	390	T	-	3.75	0.23	0.02	0.02	0.00	0.09
TOILET02.1.iv.3	28	70	F	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TOILET02.1.iv.4	37	99	T	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TOILET06.1.iv.11	294	1046	F	757.08	7.46	39.51	221.45	25.68	5.4	10.45
TOILET06.1.iv.12	321	1144	T	1633.03	5.57	18.23	74.16	12.36	1.48	0.1
TOILET07.1.iv.13	399	1491	F	-	104.47	-		599.6	92.06	141.17
TOILET07.1.iv.14	430	1608	T	-	74.77	-		265.3	23	0.22
TOILET10.1.iv.20	854	3588	T	-	-	-	-	-	-	1.31
TOILET16.1.iv.32	2132	10734	T	-	-	-	-	-	-	15.39

Table 5.3: Runtime comparison of different QBF solvers Pg. 3 (unit: seconds)

Currently, Quaffle has none of the advanced reasoning and deduction techniques employed in other state-of-the-art QBF solvers such as the pure literal rule [21], trivial truth [108], inversion of quantifiers [108], implicit unrolling [109], connected component detection [108] etc. Therefore, it is not surprising that the performance of Quaffle-naïve is not very competitive. Quaffle-BJ fares much better than Quaffle-naïve because of non-chronological backjumping. Furthermore, Quaffle-BJ is doing better on this particular benchmark suite than QuBE-BJ, perhaps because the branch heuristic employed by Quaffle (VSIDS [95]) is better suited for these examples.

For most of the benchmarks, Quaffle-CDL is much better than Quaffle-BJ. This shows the effectiveness of conflict-driven learning in a QBF setting compared with back jumping only. The only exceptions are four of the `TOILET` test cases. This may due to the noise introduced in learning that brings the search to some area that is hard to get out of. Random restarts and other techniques that can bring search out of the "valley area" may help in these cases. It should be pointed out that the effectiveness of conflict-driven learning depends on how many conflicts are encountered during the search process. In some of the benchmarks (e.g. `CHAIN`), almost all of the leaves encountered during the search are satisfying leaves. In that case, learned clauses do not contribute much compared with just back jumping.

Comparing Quaffle-CDL with Quaffle-Full, for some classes of the benchmarks such as `impl` and random 3-QBF `R3CNF`, Quaffle-Full (with satisfaction-driven implication and learning) is faster than Quaffle-CDL (with no satisfaction-driven implication and learning). For some other benchmarks such as `CHAIN` and `TOILET`, the result is not as good. For others such as BLOCKS and `logn`, the results are mixed. To get a better understanding of the performance gains and loses, some of the detailed statistics of the benchmarks are shown in Table 5.4.

From Table 5.4 it is possible to see the reasons for the performance difference between Quaffle-CDL and Quaffle-Full. In instance classes that have few satisfiable

Testcase	T/F	Quaffle-CDL			Quaffle-Full		
		#SAT Leaves	#Conf. Leaves	Run Time	#SAT Leaves	#Conf. Leaves	Run Time (s)
TOILET06.1.iv.12	F	24119	7212	18.23	17757	8414	74.16
TOILET06.1.iv.11	T	30553	11000	39.51	30419	13918	221.45
CHAIN15v.16	T	32768	43	3.15	32768	43	142.21
CHAIN16v.17	T	65536	46	6.82	65536	46	472.38
CHAIN17v.18	T	131072	49	14.85	131072	49	1794.35
impl16	T	160187	17	0.97	106	17	0.02
impl18	T	640783	19	3.88	124	19	0.02
impl20	T	2563171	21	15.51	142	21	0.02
R3CNF...3...50_8.F	F	11845	374	0.29	59	460	0.05
R3CNF...3...50_9.T	T	33224	87	0.87	35	50	0.02
lognBWLARGEA2	T	3119	11559	125.85	1937	14428	193.88
lognBWLARGEB1	F	2	601	8.26	2	609	8.18
BLOCKS4ii.6.3	F	5723	52757	367.54	98	45788	591.95

Table 5.4: Detail Statistics of CDL and Full version of Quaffle

leaves such as `logn` and `BLOCKS`, Quaffle-Full takes about the same time or just a little bit more than Quaffle-CDL because the satisfaction-driven pruning does not have much opportunity to work. For problem classes `CHAIN` and `TOILET`, though there exist many satisfying leaves, satisfaction-driven learning is not able to prune much of the search space. The reason for this is because these instances have the property that when a satisfying assignment is found, the satisfaction-induced cover set always includes all the universal literals. Because of this, the learned cubes will not be able to help prune any search space. This is just like in a SAT solver, where learned clauses with many literals are generally less effective in pruning search than short clauses. On the other hand, for instance classes `R3CNF` and `impl`, satisfaction-driven implication and learning dramatically reduce the number of satisfying leaves that need to be visited. Therefore, the total run time is reduced significantly because of the pruning.

Compared with other state-of-the-art QBF solvers, Quaffle is quite competitive in performance. Decide [108] fares very well on this particular benchmark suite, probably because the "inversion of quantifier" [108] and "implicit unrolling" [109] heuristic it employs seem to be very effective for these benchmarks. The focus in this work is to study conflict-driven and satisfaction-driven learning for QBF, and thus the experiments are focused on evaluating the gains provided by these capabilities. It would be interesting to show some representative data for the search tree size, number of implications, number of conflicts etc. for each solver. However, the source codes of other solvers are not available, and their output after each run is not quite informative. All three solvers use different implication rules, thus making direct comparison of the number of branches misleading. For example, decide [108] uses a look-ahead technique for each branch, which potentially reduces the number of decisions, but likely increases the time needed for each decision. Therefore, only the run time for each of the solvers are shown, because that is the ultimate criterion in evaluating a solver.

Learning and non-chronological backtracking can significantly prune the search space for some benchmarks. However, for some other instances, the learning and non-chronological backtracking scheme may incur overheads with little gain. The overhead of satisfaction-driven implication and learning comes mainly from two places. The first overhead is that added cubes slow down the implication process. This overhead can be reduced by an intelligent clause and cube deletion heuristic. The other overhead arises from generating the learned cubes. This overhead is tightly related to the heuristic used to generate the satisfaction-induced cover set, which in turn affects the quality of the generated cube. There are many research opportunities here to find an effective cover set without introducing a large overhead.

5.6 Related Developments

After the results described in this chapter were initially published [141, 140], the author discovered two independent papers [43] and [73] describing results similar to this work at around the same time. Here is a brief review of these other approaches. The paper by Letz [73] proposes model and lemma caching (similar to learning in this work), and dependency-directed backtracking (i.e. non-chronological backtracking in this work). However, it does not have a clean way to deal with tautology clauses and empty cubes, which is an important feature of the proposed framework. Moreover, unlike this work (as well as related results in the SAT domain), learning and non-chronological backtracking are not tightly coupled. Giunchiglia *et al.* [43] point out that "good" solutions should be represented in DNF form and use a separate set of specially marked clauses to perform the same functions as the cubes in this work do. They also use conflict-driven learning. However, their work is not resolution and consensus based. Therefore they require special treatment for the assignments (i.e. pre-fix closed) to be able to construct valid reasons.

5.7 Chapter Summary

This chapter extends the notion of learning and non-chronological backtracking for QBF solving. QBF is different from SAT in implication rules, therefore the resolution based learning causes some problem when applied directly. This work shows that learned clauses, even if they contain both a literal and its complement, obey the same implication rule as original non-tautology clauses. To accomplish satisfaction-driven learning and implication, the traditional CNF data structure is not sufficient. In the new QBF solving framework, the solver works on an Augmented CNF database instead of the traditional CNF database that many existing QBF solvers are based

on. In the new framework, the solver has an almost symmetric view of both satisfied leaves and conflicting leaves in the search tree. Implications in the new framework not only prune the search space with no solution, but also prune the search space *with* solutions. The ideas discussed in this chapter are implemented in the QBF solver Quaffle. Experiments show that just like the SAT case, learning and non-chronological backtracking can greatly help prune the search space for certain QBF instances.

Chapter 6

Conclusions and Future Work

This thesis describes the work on Boolean Satisfiability problems. Boolean satisfiability is a core problem for many applications in Electronic Design Automation (EDA) and Artificial Intelligence. This chapter draws the conclusions from the research results obtained, and looks at some future work on SAT research and related issues.

6.1 Efficient SAT Solvers

There is no argument that the most important feature of a Boolean Satisfiability Solver is its efficiency. Therefore, the majority of the research work done in the SAT community is about improving the speed of SAT solvers.

Due to its NP-Complete nature, there is no algorithm that can solve the general SAT problem in polynomial time unless P=NP, which is believed to be highly unlikely. Therefore, in the worst case, a general SAT solver based on search principles (such as most SAT solvers in existence today) needs to visit an exponential number of nodes in the search tree to determine the satisfiability of a Boolean formula. Traditionally, researchers have been concentrating on reducing the number of nodes that need to be visited in the search by improving the algorithms used in SAT solving. For example,

numerous research results have been obtained on how to prune the search space for the Davis Logemann Loveland [31] algorithm. Ideas such as a good decision strategy, powerful deduction methods, and smart learning and non-chronological backtracking have been studied and experimentally evaluated.

This thesis describes how to improve the efficiency by careful analysis and implementation of the existing algorithms. It examines the Boolean Constraint Propagation (BCP) mechanism and backtracking algorithms and shows how engineering the implementation achieves significant speed-up over current implementations. Some of the results obtained are: different BCP algorithms can significantly affect the efficiency of the overall performance of a SAT solver; a cache friendly implementation of the data structure can easily achieve 2-3x speedup; and different learning schemes may significantly affect the effectiveness of the algorithm.

For future work, there are still many things that are not well understood about the SAT solving process. For example, currently the decision strategy design is still an art instead of a science. It is very hard to explain why a certain decision strategy works better than others. There is still no way to estimate an instance's difficulty without solving it. It is still not well understood why some heuristics work with some classes of instances but not others, e.g. learning and non-chronological backtracking are useless for random instances. By getting a better understanding of the solving process, significant gains may be achieved in the efficiency of the solvers.

Improvements in SAT solving algorithms are still continuing without any sign of slowdown. Some examples of the recent developments on the algorithm side include works such as the decision heuristics proposed by Goldberg and Novikov [44], improvements in the deduction process proposed by Bacchus [7]. Many improvements can still be achieved in this area.

When dealing with specific applications, a generic SAT solver is usually not the most efficient approach. It is always desirable to implement special SAT solvers for

specific applications as seen in the case of reasoning on digital circuits (e.g. [68]). Compared with using a SAT solver as a blackbox, application specific knowledge may also help the SAT solving process when available, as shown by Strichman on bounded model checking applications [126]. There is still a lot of work to be done in this domain. As SAT solvers become widely used in various systems, it can be expected that application specific SAT solving will be an active research topic.

Some of the SAT instances encountered in practice are inherently hard for certain reasoning methods. For example, there are known results [129] on hard instances for DLL based SAT solvers (which is essentially a resolution process). Therefore, combining many reasoning methods into a single framework may be a promising direction, as shown by Kuehlmann *et al.* [68]. There are many open questions on how to combine different reasoning engines and how to make different engines cooperate with each other instead of operating independently.

6.2 Other SAT Related Issues

For many real world applications, just determining the satisfiability of a SAT instance is not enough. To make the application use a SAT solver as the core deduction and reasoning engine, the SAT solver must have additional capabilities.

This thesis describes how to certify a Boolean Satisfiability solver. When a SAT solver reports that a formula is unsatisfiable, the certifying process can perform an independent check on the result to make sure that the result is correct. This certifying process is often required by mission critical applications to detect bugs in SAT solvers. A certifying process is proposed that can work with most modern SAT solvers with minor changes. Experiments show that it takes much less time to check a proof than to actually find a proof by a SAT solver.

Another feature discussed in the thesis is to find a small unsatisfiable core from an

unsatisfiable formula in Conjunctive Normal Form. This task is desirable for debugging the instances generated from applications that *should be* satisfiable, by localizing the reason for unsatisfiability. A procedure based on utilizing the resolution tree of an unsatisfiable proof of the instance is described in the thesis and experiments show that the procedure can produce unsatisfiable cores with good quality in reasonable run time. Compared with existing approaches, the described approach can scale and work on SAT instances generated from real world applications.

There are other potentially useful features in SAT solvers. Enumerating all solutions for a satisfiable formula is one such feature. This capability is required by some applications such as model checking [91], and is also related to the problem of evaluating quantified Boolean formulas that do not have all variables quantified. Works on *counting* the number of solutions for SAT instances have been proposed in the literature [63, 30]. However, it is difficult to *produce* all the solutions because it is not trivial to represent possibly an exponential number of solutions in compact form. Another interesting feature is to determine the progress of the algorithm during the SAT solving process. This is very important for real world applications to evaluate whether to abort a solving process. There is some work on this area [5]. However, the practicality of these approaches for real time progress reporting is still unknown.

6.3 Beyond SAT: QBF and Other Research Directions

The search technique used in Boolean Satisfiability solvers can be applied to other research areas. This thesis applies the learning and non-chronological backtracking algorithm to quantified Boolean formula evaluation. The CNF formula representation used in QBF solving algorithms needs to be augmented in order to introduce the

notion of satisfiability-driven implication. The resolution based learning and assertion based non-chronological backtracking algorithms used in SAT solvers are shown to be adaptable to QBF solvers. Due to different implication rules, *long distance resolution* is needed for the learning mechanism to work. Experiments show that the heuristics developed may prune the search space greatly for certain classes of QBF instances.

An efficient QBF solver is very important for many applications. Unlike SAT, there is still significant work to be done on QBF solving to make it efficient enough for real world applications. For DLL search on QBF, many different deduction rules exist [21] and the unit implication rule may not be the most efficient one. There is virtually no research done on evaluating the effectiveness of deduction rules and branching mechanisms for QBF evaluation. A DLL search based algorithm is only one of many possible QBF solving algorithms. Other algorithms [18, 102] may also be competitive for QBF solving. How to make any of these algorithms, or some combination of them, efficient enough to solve hard practical QBF instances is still unknown.

Besides QBF, the techniques used in SAT solving can be applied to many other areas. For example, DLL search principles have been applied to 0-1 programming [82] and the covering problems [81]. It would be interesting to see the techniques developed in SAT being applied to other research fields. SAT techniques have been used for many other tasks traditionally performed by other reasoning methods. For example, recently McMillan proposed the use of SAT in unbounded model checking [91], which is traditionally a stronghold for BDD methods. Many researchers (e.g. [112]) proposed theorem provers that use SAT as the kernel engine. There is much to be explored in this area.

Bibliography

[1] DIMACS benchmark suites for Boolean satisfiability testing. Available at ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/, 1993.

[2] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, 2000.

[3] F. Aloul, M. Mneimneh, and K. Sakallah. Search-based SAT using zero-suppressed BDDs. In *Proceedings of the IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2002.

[4] F. Aloul, A. Ramani, I. Markov, and K. Sakallah. PBS: A backtrack search pseudo-Boolean solver. In *International Symposium on the Theory and Applications of Satisfiability Testing (SAT2002)*, 2002.

[5] F. Aloul and K. Sakallah. Satometer: How much have we searched? In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[6] G. Andersson, P. Bjesse, B. Cook, and Z. Hanna. A proof engine approach to solving combinational design automation problems. In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[7] Fahiem Bacchus. Enhancing Davis-Putnam with extended binary clause reasoning. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2002.

[8] Fahiem Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *International Symposium on Theory and Applications of Satisfiability Testing*, 2002.

[9] C. Berman. Circuit width, register allocation, and ordered Binary Decision Diagrams. *IEEE Transactions on Computer-Aided Design*, 10:1059–1066, 1991.

[10] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC)*, 1999.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99)*, 1999.

[12] Per Bjesse, Tim Leonard, and Abdel Mokkedem. Finding bugs in an Alpha microprocessor using satisfiability solvers. In *Proceedings of 13th Conference on Computer-Aided Verification (CAV'01)*, 2001.

[13] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Proceedings of Design Automation Conference (DAC)*, 1990.

[14] R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *International Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, 2001.

[15] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions in Computers*, 8(35):677–691, 1986.

[16] Randal E. Bryant. Binary Decision Diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1995.

[17] Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with Binary Moment Diagrams. In *Proceedings of the Design Automation Conference (DAC)*, 1995.

[18] H. Kleine Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.

[19] J. R. Burch and D. L. Dill. Automated verification of pipelined microprocessor control. In *Proceedings of the Computer-Aided Verification (CAV'94)*, 1994.

[20] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.

[21] M. Cadoli, M. Schaerf, A. Giovanardi, and M. Giovanardi. An algorithm to evaluate quantified Boolean formulae and its experimental evaluation. *Journal of Automated Reasoning*. to appear.

[22] Srimat T. Chakradhar and Vishwani D. Agrawal. A transitive closure based algorithm for test generation. In *Proceedings of the Design Automation Conference (DAC)*, 1991.

[23] P. Chatalic and L. Simon. Multi-Resolution on compressed sets of clauses. In *Twelfth International Conference on Tools with Artificial Intelligence (ICTAI'00)*, 2000.

[24] P. Chatalic and L. Simon. ZRes: The old Davis-Putnam procedure meets ZBDDs. In *17th International Conference on Automated Deduction (CADE'17)*, 2000.

[25] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[26] Stephen A. Cook and David G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17. American Mathematical Society, 1997.

[27] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an industrial setting. In *Proceedings of 13th Conference on Computer-Aided Verification(CAV'01)*, 2001.

[28] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 21–27, 1993.

[29] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 287(1):69–83, 2002.

[30] Adnan Darwiche. A compiler for deterministic, decomposable negation normal form. In *Proceedings of the National Conference in Artificial Intelligence (AAAI'02)*, 2002.

[31] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[32] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[33] Rolf Drechsler, Nicole Drechsler, and Wolfgang Gunther. Fast exact minimization of BDDs. In *Proceedings of the Design Automation Conference (DAC)*, 1998.

[34] Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. SAT versus UNSAT. In Johnson and Trick [62], pages 415–436.

[35] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, 2001.

[36] G. Finke, R. E. Burkard, and F. Rendl. Quadratic assignment problems. *Annals of Discrete Mathematics*, 31:61–82, 1987.

[37] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[38] H. Fujuwara and S. Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Transactions on Computers*, 31(6):555–560, June 1982.

[39] Micheal R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman Co., 1979.

[40] Matthew L. Ginsberg and David A. McAllester. GSAT and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, 1994.

[41] E. Giunchiglia, M. Narizzano, and A. Tacchella. Backjumping for quantified Boolean logic satisfiability. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[42] E. Giunchiglia, M. Narizzano, and A. Tacchella. Qube: a system for deciding quantified Boolean formulas satisfiability. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR)*, 2001.

[43] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified Boolean logic satisfiability. In *Proceedings of the 18th National (US) Conference on Artificial Intelligence (AAAI)*, 2002.

[44] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2002.

[45] E. I. Goldberg, M. R. Prasad, and R. K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 2001.

[46] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI'98)*, pages 431–437, Madison, Wisconsin, 1998.

[47] J. F. Groote, J. W. C. Koorn, and S. F. M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Proceedings of the 10th Annual Conference on Computer Assurance (COMPASS'95)*, pages 57–68, 1995.

[48] J. F. Groote and J. P. Warners. The propositional formula checker HeerHugo. In Ian Gent, Hans van Maaren, and Toby Walsh, editors, *SAT2000: Highlights*

of Satisfiability Research in the year 2000, Frontiers in Artificial Intelligence and Applications, pages 261–281. Kluwer Academic, 2000.

[49] Jun Gu. Local search for satisfiability SAT problem. *IEEE Transactions on Systems and Cybernetics*, 23(3):1108–1129, 1993.

[50] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (SAT) problem: A survey. In Ding-Zhu Du, Jun Gu, and Panos Pardalos, editors, *Satisfiability Problem: Theory and applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–152. American Mathematical Society, 1997.

[51] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. SAT-based image computation with application in reachability analysis. In *Proceedings of Third International Conference Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[52] D. Habet, C.M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, 2002.

[53] G. Hachtel and F. Somenzi. *Logic Sysntheiss and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[54] John Harrison. Stålmarck's algorithm as a HOL derived rule. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *9th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, 1996. Springer Verlag.

[55] Edward A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasonning*, 24(4):397–420, 2000.

[56] Edward A. Hirsch and Arist Kojevnikov. Solving Boolean satisfiability using local search guided by unit clause elimination. In *Proceedings of 7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, November 2001.

[57] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[58] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, TU Darmstadt, Germany, 1998.

[59] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, 1999.

[60] B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.

[61] R. J. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–188, 1990.

[62] D. S. Johnson and M. A. Trick, editors. *Second DIMACS implementation challenge : cliques, coloring and satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[63] R. J. Bayardo Jr. and J. D. Pehoushek. Counting models using connected components. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.

[64] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1997.

[65] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[66] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, Stockholm, Sweden, July 31-August 6 1999.

[67] Joonyoung Kim, João P. Marques Silva, Hamid Savoj, and Karem A. Sakallah. RID-GRASP: Redundancy identification and removal using GRASP. In *IEEE/ACM International Workshop on Logic Synthesis*, 1997.

[68] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. Circuit-based Boolean reasoning. In *Proceedings of the Design Automation Conference (DAC)*, June 2001.

[69] W. Kunz and D. Pradhan. Recrusive Learning: A new implication technique for efficient solutions to CAD problems: Test, verification and optimization. *IEEE Transactions on Computer-Aided Design*, 13(9):1143–1158, September 1994.

[70] A. LaMarca and R.E. Ladner. The influence of caches on the performance of heaps. *ACM Journal of Experimental Algorithmics*, 1, 1996.

[71] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[72] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, June 2001.

[73] R. Letz. Lemma, model caching in decision procedures for quantified Boolean formulas. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux2002)*, 2002.

[74] Chu-Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.

[75] Chu-Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya, Japan, August 23–29 1997.

[76] Chu-Min Li and Sylvain Grard. On the limit of branching rules for hard random unsatisfiable 3-SAT. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 98–102, Berlin, 2000.

[77] Inês Lynce. Algebraic simplification techniques for propositional satisfiability. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, March 2001.

[78] Inês Lynce, Luis Baptista, and João P. Marques Silva. Stochastic systematic search algorithms for satisfiability. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, June 2001.

[79] Inês Lynce and João P. Marques Silva. The puzzling role of simplification in propositional satisfiability. In *EPIA'01 Workshop on Constraint Satisfaction and Operational Research Techniques for Problem Solving (EPIA-CSOR)*, 2001.

[80] S. Malik, A. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using Binary Decision Diagrams in a logic synthesis environment. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1988.

[81] Vasco Manquinho and João P. Marques-Silva. On using satisfiability-based pruning techniques in covering algorithms. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 2000.

[82] Vasco Manquinho, João P. Marques-Silva, Arlindo L. Oliveira, and Karem A. Sakallah. Satisfiability-based algorithms for 0-1 integer programming. In *Proceedings of the IEEE/ACM International Workshop on Logic Synthesis (IWLS)*, 1998.

[83] Elena Marchiori and Claudio Rossi. A flipping genetic algorithm for hard 3-SAT problems. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 393–400, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[84] João P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

[85] João P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP'2000)*, September 2000.

[86] João P. Marques-Silva and Thomas Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 1999.

[87] João P. Marques-Silva and Karem A. Sakallah. Efficient and robust test generation-based timing analysis. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, May 1994.

[88] João P. Marques-Silva and Karem A. Sakallah. GRASP - a search algorithm for propositional satisfiability. *IEEE Transactions in Computers*, 48(5):506–521, May 1999.

[89] B. Mazure, L. Sas, and E. Grgoire. Tabu search for SAT. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 281–285, Providence (Rhode Island USA), 1997.

[90] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.

[91] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *Proceedings of 14th Conference on Computer-Aided Verification (CAV 2002)*. Springer Verlag, 2002.

[92] Ken L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'03)*, 2003.

[93] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of Design Automation Conference (DAC)*, 1993.

[94] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distribution of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.

[95] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC)*, June 2001.

[96] Computer Museum. Computer museum of america. http://www.computer-museum.org/, 2000.

[97] G. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: Routing complex FPGAs via search-based Boolean SAT. In *Proceedings of International Symposium on FPGAs*, Feburary 1999.

[98] K. Namjoshi. Certifying model checkers. In *Proceedings of 13th Conference on Computer-Aided Verification (CAV'01)*, 2001.

[99] E. I. Goldberg Y. Novikov. Verification of proofs of unsatisfiability for *cnf* formulas. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 2003.

[100] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 163–169, 1991.

[101] Slawomir Pilarski and Gracia Hu. SAT with partial clauses and back-leaps. In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[102] D. A. Plaisted, A. Biere, and Y. Zhu. A satisfiability procedure for quantified Boolean formulae. *Journal of Discrete Applied Mathematics*. to appear.

[103] D.A. Plaisted and S. Greenbaum. A stucture-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[104] M. R. Prasad, P. Chong, and K. Keutzer. Why is ATPG easy? In *Proceedings of the Design Automation Conference (DAC)*, New Orleans, Louisiana, United States, 1996.

[105] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[106] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and their applications. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1993.

[107] Jussi Rintanen. Constructing conditional plans by a theorem prover. *Journal of Artificial Intelligence Research*, 10:323–352, 1999.

[108] Jussi Rintanen. Improvements to the evaluation of quantified Boolean formulae. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

[109] Jussi Rintanen. Partial implicit unfolding in the Davis-Putnam procedure for quantified Boolean formulae. In *International Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2001.

[110] Jussi Rintanen. QBF benchmark suite. Available at http://ww.informatik.uni-freiburg.de/∼rintanen/qbf.html, 2002.

[111] R. Rudell. Dynamic variable ordering for Ordered Binary Decision Diagrams. In *Proceedings of the IEEE/ACM International Conference on Compute- Aided Design (ICCAD)*, 1993.

[112] S. Schulz and G. Sutcliffe. System description: Grande 1.0. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 280–284, 2002.

[113] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic Boolean programming. In *Proceedings of IJCAI-01*, 2001.

[114] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 50–54, 1997.

[115] Bart Selman, Hector Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 459–465, 1992.

[116] Julian Seward. Valgrind, a open-source memory deubugger and cache simulator. http://developer.kde.org/∼sewardj/, 2002.

[117] Y. Shang and B. W. Wah. A discrete Lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–99, January 1998.

[118] Mary Sheeran, Satnam Singh, and Gunnar Stålmark. Checking safety properties using induction and a SAT-solver. In *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000)*, 2000.

[119] Mary Sheeran and Gunnar Stålmark. A tutorial on Stålmark's proof procedure for propositional logic. *Formal Methods in System Design*, 16:23–58, January 2000.

[120] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. SAT 2002 solver competition report. http://www.satlive.org/SATCompetition/onlinereport.pdf, 2002.

[121] A. P. Sistla and E. M. Clarke. Complexity of propositional temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[122] William M. Spears. Simulated annealing for hard satisfiability problems. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 533–558. American Mathematical Society, 1993.

[123] William M. Spears. A NN algorithm for Boolean satisfiability problems. In *Proceedings of the 1996 International Conference on Neural Networks*, pages 1121–1126, 1996.

[124] G. Stålmarck. A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula. Technical report, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 1989.

[125] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design*, 15(9):1167–1176, September 1996.

[126] Ofer Strichman. Tuning SAT checkers for bounded model-checking. In *Proceedings of the Conference on Computer-Aided Verification (CAV'00)*, 2000.

[127] A. Stump and D. Dill. Faster proof checking in the Edinburgh logical framework. In *Proceedings of 18th International Conference on Automated Deduction (CADE-18)*, 2002.

[128] Prover Technology. Prover proof engines. http://www.prover.com/, 2003.

[129] Alasdair Urquhart. Hard examples for resolution. *Journal of the Association for Computing Machinery*, 34(1):209–219, 1987.

[130] Allen Van Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002.

[131] Allen Van Gelder and Yumi K. Tsuji. Satisfiability testing with more reasoning and less guessing. In Johnson and Trick [62], pages 559–586.

[132] M.N. Velev and R.E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. In *Proceedings of the Design Automation Conference (DAC)*, June 2001.

[133] L. Xiao, X. Zhang, and S. A. Kubricht. Improving memory performance of sorting algorithms. *ACM Journal of Experimental Algorithmics*, 5:1–23, 2000.

[134] H. Zhang, M.P. Bonacina, and H. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21:543–560, 1996.

[135] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

[136] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97)*, pages 272–275, 1997.

[137] Hantao Zhang and Mark Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, Dept. of Computer Science, University of Iowa, 1994.

[138] Lintao Zhang. Zchaff SAT solver. http://www.ee.princeton.edu/∼chaff/zchaff.php, 2000.

[139] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[140] Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2002.

[141] Lintao Zhang and Sharad Malik. Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In *Proceedings of 8th International Conference on Principles and Practice of Constraint Programming (CP2002)*, 2002.

[142] Lintao Zhang and Sharad Malik. Cache performance of SAT solvers: A case study for efficient implementation of algorithms. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.

[143] Lintao Zhang and Sharad Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Proceedings of Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, 2003.

[144] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the IEEE/ACM Design, Automation and Test in Europe (DATE)*, 2003.