Efficient satisfiability solver

by

Chuan Jiang

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Ting Zhang, Co-Major Professor

Wensheng Zhang, Co-Major Professor

Carl K. Chang

Iowa State University

Ames, Iowa

2014

DEDICATION

I would like to dedicate this thesis to my wife Wangyujue Hong without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and assistance during the writing of this work.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, Dr. Ting Zhang and Dr. Wensheng Zhang for their guidance, patience and support throughout this research and the writing of this thesis. Their insights and words of encouragement have often inspired me and renewed my hope for completing my graduate education. I would also like to thank Dr. Carl K. Chang for his efforts and contributions to this work.

ABSTRACT

The past few decades saw great improvements in the performance of satisfiability (SAT) solvers. In this thesis, we discuss the state-of-the-art techniques used in building an efficient SAT solver. Modern SAT solvers are mainly constituted by the following components: decision heuristics, Boolean constraint propagation, conflict analysis, restart, clause deletion and preprocessing. Various algorithms and implementations in each component will be discussed and analyzed. Then we propose a new backtracking strategy, partial backtracking, which can be easily implemented in SAT solvers. It is essentially an extension of the backtracking strategy used in most SAT solvers. With partial backtracking, the solver consecutively amends the variable assignments instead of discarding them completely so that it does not backtrack as many levels as the classic strategy does after analyzing a conflict. We implemented this strategy in our solver Nigma and the experiments show that the solver benefits from this adjustment.

CHAPTER 1. OVERVIEW

1.1 Introduction

Both theoretical research and practical applications saw the significance of the satisfiability (SAT) problem, the first and most famous NP-complete problem, leading to the research and development of SAT solvers. An efficient SAT solver is always desired even though the worst time complexity remains exponential and it seems hopeless to find an polynomial-time SAT algorithm. Unlike the absolutely random SAT instances, the instances from the real applications are believed to have some properties which can be utilized heuristically to accelerate obtaining a solution. Since their inception in the middle of 1990's, modern SAT solvers are able to solve SAT instances with millions of variables and clauses in an acceptable time and have been applied to a number of practical applications, for example, AI planning [27], scheduling [48][22], electronic design automation (EDA) [45], software verification [24] and model checking [10][8].

In practice, SAT solvers are utilized as a black-box tool. Figure 1.1 illustrates how to apply SAT-based solving techniques. First, the original problem is transformed into a satisfiability problem in the form of Boolean formula in polynomial time. Then the SAT solver is used to determine the satisfability. If satisfiable, the solver produces an assignment that makes the formula evaluate to TRUE. It is possible that the solver runs several times, refining the Boolean formula iteratively with the assignment obtained in each iteration. The final answer is transformed in reverse to give a solution for the original problem. The transformation to SAT generally leads to a substantial increase in the problem representation. However, for many combinational search and reasoning tasks, the transformation followed by the use of a modern SAT solver often proves more effective and efficient than a custom search engine running on the original problem formulation.

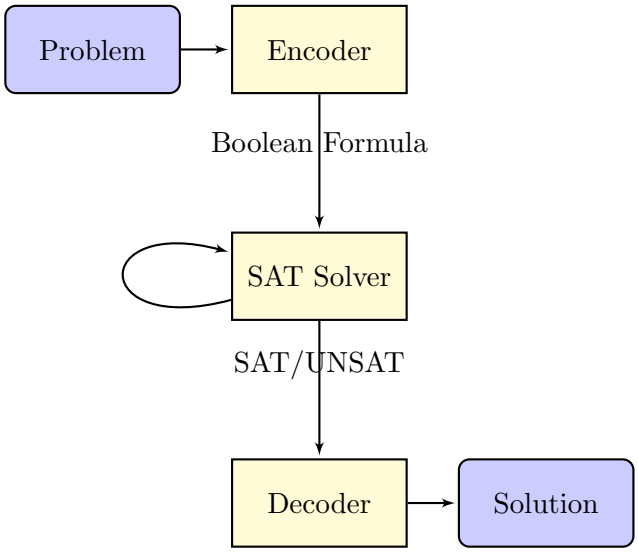


Figure 1.1: Problem solving process with SAT solver

This thesis surveys the recent improvements of SAT solvers. We mainly focus on conflict-driven clause learning (CDCL) solvers [34], whose organization is primarily inspired by Davis-Putnam-Logemann-Loveland (DPLL) solvers [12][11]. Then a new backtracking strategy, partial backtracking [26], is proposed to reduce the negative influence of repeated propagation. With partial backtracking, the solver consecutively amends the variable assignments instead of discarding them completely so that it does not backtrack as many levels as the classic strategy does after analyzing a conflict. We implemented this strategy in our solver Nigma and the experiments show that the solver benefits from this adjustment.

1.2 Organization

This thesis is organized as follows. Chapter 2 is a brief review of propositional logic and the satisfiability problem. Chapter 3 gives an overview of SAT solvers. Details and implementations of key techniques are covered in Chapter 4. We introduce our contribution, partial backtracking, and show that SAT solver benefits from it in Chapter 5. Chapter 6 concludes this thesis.

CHAPTER 2. SATISFIABILITY

2.1 Introduction

This chapter provides a brief review of propositional logic and the satisfiablility problem. The terminology and notation introduced will be used throughout the thesis.

2.2 Propositional Logic

In mathematical logic, *propositional logic* is a formal system that is concerned with propositions and their interrelationships. The basic element in propositional logic is *Boolean variable*, or *propositional variable*, whose value can only be TRUE or FALSE. The *Boolean formula*, or *propositional formula*, is a combination of Boolean variables and connectives including an unary operator *negation* ($\neg$) and four binary operators *conjunction* ($\wedge$), *disjunction* ($\vee$), *implication* ($\Rightarrow$) and *equivalence* ($\Leftrightarrow$). Furthermore, we introduce two concepts useful in the satisfiability problem.

Definition 1 *A literal is either a Boolean variable x or its negation $\neg x$.*

Definition 2 *A clause is a disjunction of literals.*

We say a variable or literal is *free* if it is unassigned. A clause is *satisfied* if it evaluates to TRUE, that is, at least one of its literals is assigned TRUE. A clause is *unsatisfied* if all its literals are assigned FALSE. A clause is *unit* if all its literals but one are assigned FALSE, and the remaining literal is free.

Definition 3 *Given a set of Boolean variables $X = \{x_1, \cdots, x_n\}$, an assignment is a function $v : X \rightarrow \{TRUE, FALSE, UNASSIGNED\}$.*

Given an assignment v, if all variables are assigned either TRUE or FALSE, then v is referred to as a *complete assignment*. Otherwise it is a *partial assignment*. It is easy to see that there exist 2^n complete assignments for a set of n Boolean variables. We call the mapping from a Boolean variable to a Boolean value a *variable assignment*.

2.3 Satisfiability Problem

Definition 4 *A Boolean formula is satisfiable if there exists a complete assignment such that the formula evaluates to TRUE.*

The satisfiability (SAT) problem is to determine if a Boolean formula is satisfiable or not. In practice, one is not only interested in the answer "yes/no", but also finding an actual assignment satisfying the formula if there exists one, or an unsatisfiability proof if not. SAT is the most famous and studied NP-complete problem because lots of problems can be transformed into it in polynomial time. Unfortunately, researchers have not found any polynomial-time SAT algorithm, and it seems hopeless that there exists one. Theoretically, a Boolean formula cannot be proved unsatisfiable until all the 2^n possibilities have been tried.

Definition 5 *A Boolean formula is in conjunction normal form (CNF) if it is a conjunction of clauses.*

Lemma 1 *The set of connectives* $\{\neg, \wedge, \vee\}$ *is adequate in propositional logic.*

According to Lemma 1, any Boolean formula can be transformed into CNF. We take CNF as the standard form of the SAT problem since many problems are naturally expressed as a conjunction of relatively simple constraints. For convenience, a CNF formula can also be viewed as a set of clauses, which is referred to as *clause database*, and a clause as a set of literals.

CHAPTER 3. SATISFIABILITY SOLVER

3.1 Introduction

Although there seems no efficient algorithm to solve the SAT problem, a SAT utility is always desired for theoretical and practical purposes, leading to the research and development of SAT solvers. Researchers believe that the SAT instances from the real applications have backdoors or special structures, which can be utilized heuristically. The past decades saw great improvements in the performance of SAT solvers. Modern SAT solvers are able to solve SAT instances with millions of variables and clauses in an acceptable time, and they have found applications in AI planning [27], scheduling [48][22], electronic design automation (EDA) [45], software verification [24], model checking [10][8], etc.

The first significant breakthrough in the research of SAT solver is Davis-Putnam-Logemann-Loveland (DPLL) algorithm [12][11], which inspires the standard organization of modern SAT solvers. Then GRASP [34] introduces learning and non-chronological backtracking into DPLL solvers and prompts the research on conflict-driven clause learning (CDCL) solvers in recent decades. Almost all the state-of-the-art SAT solvers follows CDCL, including MiniSat [14], Lingeling [5], Glucose [1], CryptoMiniSat [41], Riss [32], etc. In this thesis, we will not discuss about other solvers based lookahead[20] or local search [40][39].

This chapter is organizaed as follows. Section 3.2 outlines the standard organization of DPLL solvers. Section 3.3 extends DPLL and explains CDCL solvers.

3.2 DPLL

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm was introduced by Martin Davis, Hilary Putnam, George Logemann and Donald Loveland in 1962 [11], and is a refinement of

the earlier David-Putnam algorithm [12]. It is essentially a complete depth-first backtracking search algorithm, efficiently pruning the search space based on falsified clauses. Algorithm 1 shows the standard organization of a DPLL solver.

Algorithm 1 A pseudo-code of DPLL solver

```
1: while true do
2:   if !decide() then
3:     return SAT
4:   end if
5:   if !booleanConstraintPropagate() then
6:     if !resolveConflict() then
7:       return UNSAT
8:     end if
9:   end if
10: end while
```

At the beginning of each iteration, the function *decide*() selects a free variable and assigns it some value heuristically. This variable assignment is called a *decision* and pushed into a stack. A *decision level* is associated with each decision to denote its depth in that stack. If all the variables have values, *decide*() returns FALSE to indicate that the SAT instance is satisfiable.

Then *booleanConstraintPropagate*() is invoked. This process utilizes *unit propagation rule*: a unit clause asserts that the sole free literal must be assigned TRUE for the clause to be satisfied. We call that assertion an *implication*, written as $l@dl$, indicating that the literal l is implied to be TRUE at the decision level dl, and the unit clause the *antecedent clause* of the implication, indicating that the clause is the reason of the implication. *Boolean constraint propagation* (BCP) is the iterative process of searching for unit clauses and obtaining implications until reaching a fixed point or encountering a conflict, that is, finding an unsatisfied clause. We call that clause a *conflicting clause*. The function *booleanConstraintPropagate*() returns FALSE to indicate that the solver encounters a conflict during BCP.

If BCP terminates without a conflict, the solver makes another decision and propagates it. Otherwise, the solver flips the most recent decision if it hasn't been flipped, or revokes decisions until an unflipped decision is reached, in $resolveConflict()$. This kind of backtracking is called *chronological backtracking*. If all the decisions have been flipped and a conflict occurs, $resolveConflict()$ returns FALSE to indicate that the SAT instance is unsatisfiable.

3.3 Conflict-Driven Clause Learning

João P. Marques-Silva and Karem A. Sakallah extended DPLL with learning and non-chronlogical backtracking and introduced Generic seaRch Algorithm for the Satisfiability Problem (GRASP) [34] , prompting the research and development of conflict-driven clause learning (CDCL) solvers. The organization of CDCL solvers follows Algorithm 1, except a different implementation of $resolveConflict()$.

CDCL solvers treat each conflict an opportunity to learn more about the SAT instance. Each time a conflict is identified, the solver extracts the reason as a clause by some learning scheme and adds it into the clause database to avoid recurrence of that conflict and help prune the search space in the future. This process is called *conflict analysis* or *learning*, and the newly added clause is called a *learnt clause*. Guided by the learnt clause, the solver then backtracks to some earlier level which may not be the one with the most recent unflipped decision, potentially pruning a larger portion of the search space. This kind of backtracking is called *non-chronological backtracking*.

The search with learning is still complete as the learnt clause can be inferred from the existing clauses. If c is a new clause learnt from the CNF formula Σ, then Σ is satisfiable if and only if $\Sigma \bigcup \{c\}$ is satisfiable. Moreover, the non-chronological backtracking also does not affect soundness or completeness, since the backtracking information is obtained from each new learnt clause.

We note that learning does not change the fact that the worst case time complexity remains exponential in terms of the number of variables. However, in the case of some classes of real applications, a good implementation shows an acceptable time complexity when combined with appropriate heuristics.

CHAPTER 4. KEY TECHNIQUES

4.1 Introduction

In this chapter, we discuss the key techniques in building an efficient SAT solver and survey various implementations. From Algorithm 1, we know that the basic techniques to determine a SAT solver's performance consists of decision heuristics, Boolean constraint propagation and learning scheme. However, the typical algorithm shown in Algorithm 1 does not take into account a few often used techniques, namely restart, clause deletion and preprocessing, which will be also covered in this chapter.

4.2 Decision Heuristics

Decision heuristic has a significant impact on the performance of the solver. Even for the same basic solver structure, different decision heuristics may produce search trees with drastically different sizes. If the solver is "lucky" to choose a "good" branch at the early stage, lots of time spent in fixing its faults can be saved.

Making a decision requires two steps. First, select a free variable. Second, assign the variable a selected value. We introduce two kinds of variable selecting heuristics in Section 4.2.1 and Section 4.2.2, and a commonly-used value selecting heuristic in Section 4.2.3.

4.2.1 State-dependent Heuristics

The early decision heuristics made use of the information available from the data structures. Some examples are Maximum Occurrence in clauses of Minimum Size (MOMS) and Dynamic Largest Individual Sum (DLIS). The common point is making decisions that make the resulting formula as "simple" as possible. However, these strategies are state-dependent because different

variable assignments will give different numbers of occurrence of literals or variables, and these numbers must be updated once a variable is assigned or unassigned, making it expensive to maintain.

4.2.2 VSIDS

In order to release the solver from the high price of maintaining the occurrence, a state-independent decision heuristic, Variable State Independent Decaying Sum (VSIDS), was proposed in Chaff [35]. For each variable, VSIDS keeps a score and increases it once this variable involves in a conflict. Thus the score can be used to evaluate how active a variable is in the CNF formula: the higher score a variable has, the more active it is. The solver always selects a free variable with highest score. In such a manner, the solver gives priority to assigning active variables. Meanwhile, all scores are halved periodically so that the solver focuses on active variables in recent conflicts. VSDIS proves more effective in CDCL solvers than the state-dependent heuristics, indicating that emphasizing a kind of locality is more favorable than simplifying the formula.

VSIDS has several variants. For example, the solver Berkmin [16] also measures clauses' age and activity for deciding the next assgined variable, and the solver siege [38] gives priority to assigning variables on recently added clauses.

4.2.3 Phase Saving

Another problem in making a decision is which Boolean value to be assigned to the selected variable. SAT solvers benefit from setting the initial value of variables FALSE. It can be explained by the phenomenon that most variables are assigned FALSE in real-life satisfiable instances.

After that, a lot of solvers select Boolean value by a method called phase saving [37], which is inspired by the following observation: the variable assignments discarded by non-chronological backtracking may contain solutions of sub-problems. The solver has to rediscover them in a later stage, as it did not save the solutions previously found. Phase saving is a low-overhead caching technique that always assigns the free variable its last value. In this manner, the

solutions of sub-problems can be reconstructed in a different order and the solver enjoys a decrease of work repetition.

4.3 Boolean Constraint Propagation

Experiments show that for most SAT instances, a major portion (70% $\sim$ 90%) of the solvers' runtime is spent in the process of Boolean constraint propagation (BCP). Therefore, implementing an efficient BCP is key to any SAT solver. We discuss two kinds of BCP implementations in Section 4.3.1 and Section 4.3.2, respectively. Note that the second one has become a standard method of SAT solvers.

4.3.1 Occurrence List

The early BCP implementation maintains an occurrence list for each literal. An occurrence list contains the references of clauses in which the corresponding literal occurs. Once a variable is assigned, the solver examines the clauses in its FALSE literal's list one by one to see if the clause becomes unit or unsatisfied. This naive method is computationally expensive, requiring the solver to examine lots of clauses that are not unit or satisfied. Meanwhile, the memory consumption is almost doubled. Assuming there are n clauses and the average number of literals in a clause is m, the solver needs additional mn space to maintains the occurrence lists.

Another implementation is to maintain the number of TRUE literals and the number of FALSE literals for each clause in addition to the occurrence lists. However, it still cannot eliminate excessive examination.

4.3.2 Two Watched Literals

The solver Chaff [35] proposed a lazy data structure, the watched literals, to accelerate BCP. The idea exploits the fact that a clause with n literals is possible to be unit or unsatisfied only after its $n-1$ literals are assigned FALSE. In other words, instead of visiting every clause containing a literal which is assigned FALSE recently, we only need to visit the clauses whose number of FALSE literals goes from $n-2$ to $n-1$.

Algorithm 2 $Propagate(l@dl)$

```
1: wl_1 ← ¬l
2: for all clause c where wl_1 is watched do
3:    Search for a non-FALSE unwatched literal l' in c
4:    if Exists l' then
5:       Unwatch wl_1
6:       Watch l'
7:    else
8:       wl_2 ← the other watched literal in c
9:       if wl_2 is FALSE then
10:         ImplicationQueue.Clear()
11:         ConflictAnalysis()
12:         return
13:      else if wl_2 is TRUE then
14:         continue
15:      else
16:         ImplicationQueue.Push(wl_2@dl_curr) {dl_curr is the current level}
17:      end if
18:   end if
19: end for
```

Algorithm 3 $BCP()$

```
1: while ImplicationQueue is not empty do
2:    l@dl ← ImplicationQueue.Pop()
3:    Propagate(l@dl)
4: end while
```

To exploit this fact, the solver picks any two literals that are not assigned FALSE to watch for each clause. Thus, the solver guarantees that there cannot be more than $n-2$ FALSE literals in this clause until one of its watched literals is assigned FALSE, which is the only necessary moment when the solver needs to visit this clause.

For each clause except the satisfied ones, the solver must maintain the property that both watched literals are not assigned FALSE. When the solver visits a clause and finds that it is not unit or unsatisfied, the solver stops watching the FALSE watched literal and watches an unwatched non-FALSE literal instead.

Algorithm 2 shows the propagation of an implication with the method two watched literals, and Algorithm 3 shows the iterative process of propagation.

Assuming there are n clauses, this method works with additional $2n$ space, which is much

smaller than the naive methods. Besides, it incurs no overhead in backtracking. After backtracking, the property that both watched literals are not assigned FALSE for each clause that is not satisfied still holds, and no extra work is needed to maintain it.

4.4 Conflict Analysis

Conflict analysis, or learning, is the most important feature that distinguishes CDCL solvers from DPLL solvers. The modern solvers treat each conflict as an opportunity to learn more about the SAT formula. Meanwhile, the knowledge learnt from conflicts is also used to guide backtracking. We will explain how the solver gains knowledge from a conflict in Section 4.4.1. A technique to refine the knowledge will be discussed in Section 4.4.2.

4.4.1 Learning Scheme

The basic principle of conflict analysis is to identify the reason of a conflict, express its negation as a clause and add this clause, which is called *learnt clause*, into the clause database. Additionally, the learnt clause is desired to have only one literal at the current decision level so that it can be used to guide backtracking.

The solver identifies the reason of a conflict with the help of a directed acyclic graph (DAG) called *implication graph*. A typical implication graph is illustrated in Figure 4.1. Each vertex represents a variable assignment (e.g. $\neg x_2$@10 indicates that x_2 is assigned FALSE at the level 10). The incident edges of each vertex start from the reasons of that variable assignment (e.g. $\neg x_2$ is implied by x_4 and x_7). A decision vertex has no incident edges. A conflict occurs when there are vertices for both TRUE and FALSE assignments of a variable, which are called *conflict vertices*.

Identifying the reason of a conflict is essentially finding a bipartition consisting of the reason side and the conflict side such that the reason side contains all decision vertices and the vertices whose decison level is lower than the current level, and the conflict side contains all conflict vertices. Given an implication graph, it is obvious that there might exist multiple bipartitions satisfying these requirements. The vertices which has an edge going through the cut cause the conflict jointly.

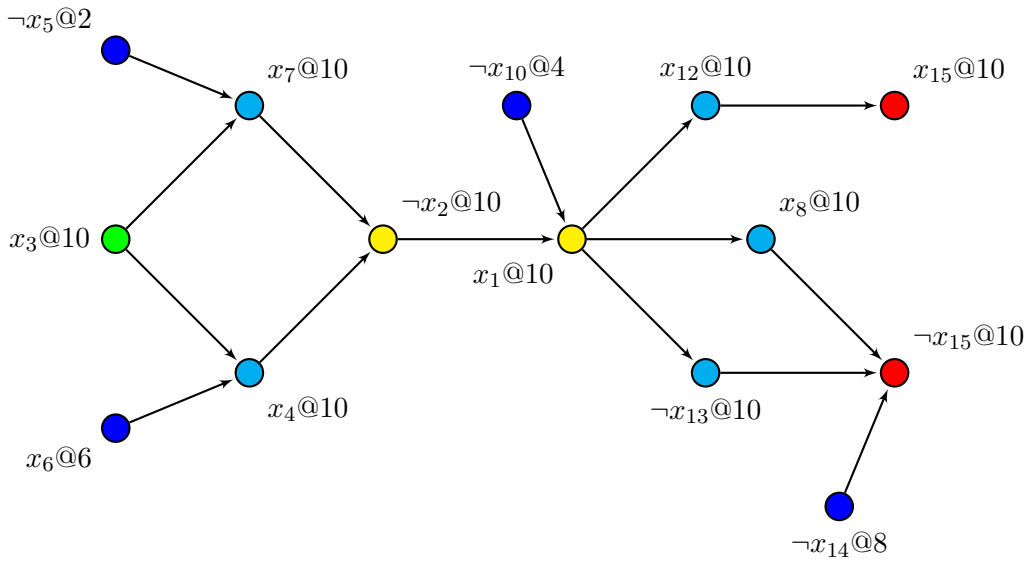


Figure 4.1: A typical implication graph

In an implication graph, there are some vertices we pay special attention to, such as Unique Implication Point [34][49].

Definition 6 *Given an implication graph, a vertex v is a Unique Implication Point (UIP) at the decision level dl iff any path from the decision vertex at dl to the conflicting vertices needs to go through v.*

In Figure 4.1, $x_3@10$, $\neg x_2@10$ and $x_1@10$ are UIPs. We order UIPs from the conflict vertices and the decision vertex is always the last UIP.

Different learning schemes vary in which cut is selected to identify the reason of a conflict. Experiments show that 1-UIP [49] achieves the best performance among all known learning schemes. 1-UIP means the implication graph is partitioned before the first UIP. For the case in Figure 4.1, according to 1-UIP, the reason of the conflict is $(x_1 \wedge \neg x_{14})$ and the learnt clause is its negation, $(\neg x_1 \vee x_{14})$. The intuition behind 1-UIP is to find a reason closest to the conflict. In practice, we need not to construct the entire implication graph. For 1-UIP, the construction, generally starting from the conflict vertices, terminates earlier than other schemes, such as 2-UIP and Last UIP.

GRASP's learning scheme [34] tries to learn as much as possible from a conflict, that is, adds more than one clause each time it meets a conflict. But it incurs more serious exponential explosion of the clause database and the experiments show that the loss outweighs the gain [49].

4.4.2 Learnt Clause Minimization

It is possible that the learnt clause contains some redundant knowledge. Niklas Sörensson and Armin Biere proposed an algorithm to minimize the learnt clause by removing additional literals [44]. According to the algorithm, the solver marks all the literals in the learnt clause obtained by 1-UIP. The implied literals in the learnt clause are candidates for removal. For each candidate, the solver constructs the implication graph starting from its antecedent literals and ending at marked literals or decisions. If the construction always ends at marked literals then the candidate can be safely removed. Since the minimized learnt clause always subsumes the original one (a clause c_1 subsumes c_2 if $c_1 \subseteq c_2$), this technique not only saves space, but also reduces the search space.

4.5 Restart

As has been argued in [19] and [18], DPLL procedure is identified to have the heavy-tailed behavior, which is characterized by a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before [17]. Even if an instance is easy to satisfy or to refute, the solver may get stuck in a complex part of the search space. A randomization technique, namely restart, was proposed to eliminate the negative influence of that behavior. With restart, the solver periodically discards the entire assignment trail and starts over again, retaining all the clauses learnt so far. Due to the wide adoption of VSIDS and phase saving, more and more researchers view each restart as a rearrangement of variable dependencies and agree that SAT solvers benefit from rapid restart. The core of a restart policy is to determine when the solver performs a restart. Two kinds of scheduling, static scheduling and dynamic scheduling, will be discussed in Section 4.5.1 and Section 4.5.2, respectively.

4.5.1 Static Scheduling

The simplest static restart scheduling is to follow a geometric series, which is used in MiniSat 1.4 [15]. At the beginning, the solver sets the limit i for the conflict interval to a initial value (e.g. 100). Whenever the number of conflicts since the last restart reaches i, the solver performs a restart and i increases by a ratio (e.g. 1.5).

In [23] Luby series is demonstrated to be very efficient. This series follows a slow but exponentially increasing law like $\{1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, \cdots\}$ and is proved to be optimal for the search algorithms without information about the search space [28]. A unit is selected to generate the series of limits for the conflict interval (e.g. suppose the unit is 100, the resulting series is $\{100, 100, 200, 100, 100, 200, 400, 100, 100, 200, 400, 800, \cdots\}$).

Then a nested restart scheduling, which nests two series, was proposed and implemented in the solver PicoSat [7]. The solver maintains an inner limit i and an outer limit o. The outer series bounds the inner series. Whenever the number of conflicts since the last restart reaches i, the solver performs a restart and i is set to the next value of the inner series. Whenever i reaches o, i is reset to the initial value of the inner series and o is set to the next value of the outer series.

Figure 4.2 shows the three kinds of series. In practice, Luby scheduling and nested scheduling are the most widely-adopted static ones for SAT solvers, performing better than geometric scheduling.

4.5.2 Dynamic Scheduling

Unlike static scheduling, dynamic scheduling determines when to perform a restart by analyzing the data collected during execution. The solver PicoSat keeps an eye on the frequency of flips [6]. We consider it as a *flip* if a variable is forced to be assigned the opposite of its last value. If the frequency of flips is small, then the SAT solver literally does not move much in the search space, and this may be a good time to restart. On the other hand if many flips occurs recently, the next restart will be deferred.

In Glucose 2.2 [3] when to perform a restart is determined by analyzing the quality of the

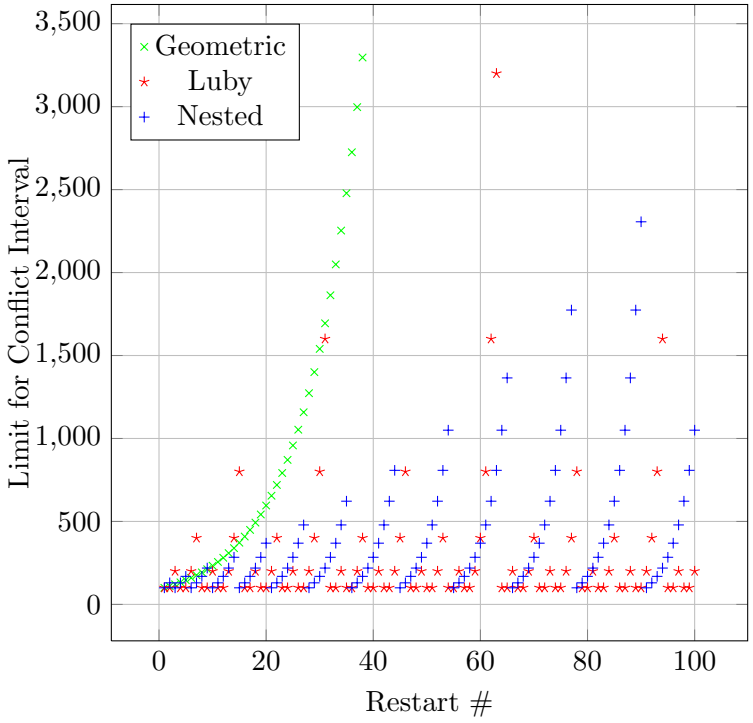


Figure 4.2: Static scheduling

recent learnt clauses, which is measured by LBD (see Section 4.6.2), instead of the number of conflicts. The solver maintains a bounded queue of the LBDs of the recent learnt clauses and performs a restart if the average of the LBDs in the queue is relatively large. Besides, the solver delays a restart if the number of assigned variables is significantly above the average during the window of recent conflicts, taking into account the possibility that the solver is approaching to proving the satisfiability of the instance.

4.6 Clause Deletion

Learning is a double-edged sword. Since the knowledge gained from conflicts is represented as clauses and added into the formula continuously, the formula would grow exponentially and then deteriorates the core of CDCL solvers: their BCP performance drops down, making some problems yet much harder to solve. Thus the solver has to delete some learnt clauses periodically in order to keep BCP efficient. However, deleting too many clauses risks discarding useful knowledge and breaking the overall learning benefit. Lots of effort has been put into

finding heuristics to accurately measure the quality of a learnt clause.

4.6.1 Activity Heuristics

Inspired by VSIDS, MiniSat 1.4 [15] keeps an activity score for each clause and increases that score each time the clause is used in conflict analysis. The learnt clauses with low activity score are seen inactive and removed periodically.

4.6.2 LBD Heuristics

Gilles Audemard and Laurent Simon observed that decision level is decreasing in most CDCL solvers on most industrial benchmarks and introduced a new property Literal Blocks Distance to measure the quality of a learnt clause [2].

Definition 7 *Given a learnt clause, its literals can be partitioned into sets where all literals are assigned at the same decision level. The number of sets is the Literal Blocks Distance (LBD) of that clause.*

For each learnt clause, its LBD is immediately computed when it is learnt by conflict analysis, and updated when it contributes to an implication during propagation and the new value is smaller than the old one. It has been shown that the learnt clauses with a small LBD are important for conflict analysis, which can partly explain the efficiency of 1-UIP by its ability to produce such clauses. During deleting clauses, the solver aggressively removes the clauses with a large LBD. The clauses with LBD=2 are never removed because they assign all their literals at a single level after backtracking.

4.7 Preprocessing

Although the encoding methods that transform real-life problems into SAT instances have a significant impact on the performance of SAT solvers and vary by problem classes, preprocessing techniques serve as a universal way to simplify SAT instances before search, regardless what domain the instances stem from. There are two benefits of preprocessing SAT instances. First, reducing the size of SAT instances by removing redundant knowledge generally enhances the

robustness of SAT solvers. Second, special kinds of structures can be identified and handled efficiently before search.

The preprocessed instance is required to be assignment-preserving or satisfiability-preserving to the original one. An instance is *assignment-preserving* if given a complete assignment, it always evaluates to the same Boolean value with the original one. An instance is *satisfiability-preserving* if it is satisfied with a complete assignment satisfying the original one. However, the complete assignment satisfying the preprocessed instance may not satisfy the original one. But by combining the knowledge removed during preprocessing, it can be used to construct a complete one satisfying the original instance. Among the preprocessing techniques to be discussed, failed literal probing and vivification preserve assignment while variable elimination and blocked clause elimination preserve satisfiability.

4.7.1 Variable Elimination

In theory, there is no explicit relationship between the scale and the difficulty of a CNF formula. A very large formula may be easy to solve and a small one hard. However, in practice, it is often observed that the runtime of a SAT solver is very much related to the size of the input formula, especially when the formulas stem from the same set of problems.

Variable elimination [13][46] is a technique that reduces the number of variables while does not increase the number of clauses. Given two clauses $c_1 = (x \vee a_1 \vee \cdots \vee a_n)$ and $c_2 = (\neg x \vee b_1 \vee \cdots \vee b_m)$, the implied clause $c = (a_1 \vee \cdots \vee a_n \vee b_1 \vee \cdots \vee b_m)$ is called the *resolvent* of c_1 and c_2 on x. We write $c = c_1 \bigotimes c_2$. Let S_x denote the set of clauses containing x and $S_{\neg x}$ the set of clause containing $\neg x$. Then we define $S = S_x \bigotimes S_{\neg x} = \{c_x \bigotimes c_{\neg x} | c_x \in S_x, c_{\neg x} \in S_{\neg x}\}$. Note that if we replaces the original $S_x \cup S_{\neg x}$ with S, the resulting formula does not contain the variable x but the satisfiability is preserved. Such a replacement takes places only when $|S| < |S_x \cup S_{\neg x}|$ to keep the number of clauses decreasing. In practice, some heuristics are exploited to select variable candidates to be eliminated in order to avoid spending too much time on failed attempts.

If the original formula is satisfiable, solving a simplified formula can only give a partial assignment. Thus the clauses removed during variable elimination must be stored and used to

get the eliminated variables assigned and produce a complete assignment.

4.7.2 Failed Literal Probing

Definition 8 *Given a CNF formula Σ, the literal l is a failed literal if $\Sigma \wedge \{l\}$ always leads to a contradiction.*

Finding a failed literal helps to discard a half of search space because Σ is possible to be satisfiable only when $\neg l$ is implied. It is used in failed literal probing [30] which is implemented as follows: given a variable x, propagate x and $\neg x$ at the level 0, respectively. If any try leads to a contradiction, we get a failed literal and its negation is implied. If both lead to a contradiction, we immediately conclude that the formula is unsatisfiable.

4.7.3 Vivification

Vivification[36] performs an incomplete redundancy check on each original clause through unit propagation, leading to either a sub-clause or a new relevant one generated by the learning scheme. In this way, the original clause could be substituted by the more constrained one and the CNF formula could be vivified, namely, made easier to solve.

Given a clause $c = (l_1 \vee l_2 \vee \cdots \vee l_n)$, the preprocessor assigns its literals FALSE one by one. Suppose the literals get assigned in the order $(\neg l_1, \neg l_2, \cdots, \neg l_n)$. While propagating l_i $(1 \leq i \leq n-1)$:

If a conflict is identified, we can immediately conclude the clause $c' = (l_1 \vee \cdots \vee l_i)$, which subsumes c. By performing conflict analysis, c' is possible to be shortened again.

If one of the remaining literals l_j $(i+1 \leq j \leq n)$ is assigned, there are two possible cases: If l_j is assigned TRUE, we get the clause $c' = (l_1 \vee \cdots l_i \vee l_j)$, which subsumes c. Otherwise, we get the clause $(l_1 \vee \cdots \vee l_i \vee \neg l_j)$. Further, a clause c' subsuming c can be produced as resolvent, $c' = (l_1 \vee \cdots \vee l_n) \bigotimes (l_1 \vee \cdots \vee l_i \vee \neg l_j) = (l_1 \vee \cdots \vee l_{j-1} \vee l_{j+1} \vee \cdots \vee l_n)$.

Accordingly, by assigning the literals in a clause FALSE iteratively, a sub-clause could be produced, removing the original clause from the CNF formula safely.

4.7.4 Blocked Clause Elimination

Blocked clause elimination[25] simplifies the original CNF formula by removing so called blocked clauses.

Definition 9 *A literal l in a clause c of a CNF formula Σ blocks c if for any clause $c' \in \Sigma$ with $\neg l \in c'$, the resolvent $c \bigotimes c'$ on l is a tautology.*

Definition 10 *A clause is blocked if it has a literal that blocks it.*

It can be proved that removing blocked clauses preserves satisfiability. Moreover, the result of blocked clause elimination is independent of the order in which blocked clauses are removed. In practice, similar with variable elimination, the removed blocked clauses are retained to produce a complete assignment if the simplified formula is satisfiable. And a heuristic cut-off limit is set to avoid finding blocked clauses for the negation of a literal with a large number of occurrences.

4.8 Additional Techniques

Researchers have realized that SAT instances from specific domains may have special features and proposed techniques which are especially useful to solve these instances. For example, observing that a relatively large amount of XOR constraints exist in the instances from cryptographical applications, Mate Soos exploits on-the-fly Gaussian elimination and gains a speedup [43][42].

Other implementation tricks are designing cache conscious data structures [9], improving resource usage [21][33], etc.

CHAPTER 5. PARTIAL BACKTRACKING

5.1 Introduction

Most modern SAT solvers are based on conflict-driven clause learning (CDCL). As a basic technique of CDCL solvers, *backtracking* helps the solver jump out of a local search space where no solution could ever be found [34]. In CDCL solvers, backtracking is non-chronological and guided by conflict analysis to determine how far the solver would jump back. The first non-chronological backtracking strategy was introduced in GRASP [34]. When GRASP meets a conflict, it keeps the current level and flips the value of the most recent decision variable. Backtracking only occurs if the flipping still leads to a conflict. Later, *random backtracking* was proposed to introduce randomness into selecting the backtracking level [29; 31]. Essentially, the learnt clause is used for randomly deciding which variable is to be flipped. Nowadays, most solvers utilize a non-randomized backtracking strategy [35], which is referred to as *classic backtracking* in this chapter. This strategy is more aggressive than that used in GRASP, since backtracking is always carried out after each conflict, making the resulting assignment trail always look like the one obtained when the learnt clause has already been included in the formula.

No matter what kind of backtracking a solver takes, it is observed that sometimes the solver backtracks quite far, which is almost equivalent to a restart. However, due to the wide adoption of VSIDS [35] and phase saving [37], the solver may make similar decisions as the ones before backtracking and hence repeat some propagations. In this paper, we present a new backtracking strategy, referred to as *partial backtracking* [26]. We implemented this strategy in our solver Nigma. Using this strategy, Nigma amends the variable assignments between the conflicting level and the assertion level instead of discarding them completely. Nigma still

backtracks after each conflict, but it does not have to backtrack as many levels as those solvers using classic backtracking. Our experiments show that Nigma backtracks 10% $\sim$ 60% fewer levels than the version with classic backtracking.

This chapter is organized as follows. Section 5.2 analyzes the classic backtracking strategy and the phenomenon of repeated propagation. Section 5.3 presents the implementation details of the partial backtracking strategy. Several optimizations on the implementation are discussed in Section 5.4. Section 5.5 presents the experiment results, showing the performance of our solver Nigma is improved after adopting the partial backtracking strategy. Section 5.6 concludes with some discussion on the future work.

5.2 Classic Backtracking

In this section, we present the classic backtracking and identify the phenomenon of *repeated propagation*.

According to the classic backtracking, the solver resolves conflicts by backtracking to the *assertion level* dl_{asrt}, which is the second highest level among the literals in the learnt clause (we say a level dl_1 is higher than dl_2 if $dl_1 > dl_2$), and hence erasing all the variable assignments between dl_{asrt} and the *conflicting level* dl_{conf}, which is the level where the conflict occurs. After backtracking, the learnt clause becomes unit and the solver invokes BCP. This kind of backtracking unavoidably discards all the propagations between dl_{asrt} and dl_{conf}.

Peter van der Tak et al. observed that CDCL solvers may reassign the same variables to the same Boolean values after a restart, and proposed the *partial restart* strategy [47]. One important reason of reassignments is the wide adoption of VSIDS [35] and phase saving [37]. We observed that backtracking exhibits a similar phenomenon, which we refer to as *repeated propagation* (note that a restart is a special form of backtracking). We give an example to illustrate this phenomenon.

Consider the clauses and variable assignments in Figure 5.1a and Figure 5.1b. Since the solver tends to select the most active free variables and their last values as decisions, we have the resulting assignment trail shown in Figure 5.1c. Then the solver encounters a conflict while propagating x_8 at the level 5 (the conflicting clause is framed in Figure 5.1a). The clause

$\neg x_1 \vee x_2$
$\neg x_3 \vee \neg x_4$
$\neg x_1 \vee x_4 \vee x_5 \vee x_6$
$x_5 \vee x_{13}$
$\neg x_7 \vee x_8$
$\neg x_7 \vee x_9$
$\neg x_2 \vee \neg x_8 \vee x_{10}$
$\boxed{\neg x_8 \vee \neg x_9 \vee \neg x_{10}}$
$x_4 \vee x_7 \vee \neg x_{11}$
$x_7 \vee x_{11} \vee x_{12}$
$x_6 \vee x_{11}$

(a) Clauses

Variable	Activity Score	Last Value
x_1	10	TRUE
x_3	8.1	TRUE
x_2	7.2	TRUE
x_5	6.4	FALSE
x_{12}	6	FALSE
x_7	5.5	TRUE
x_6	3.7	FALSE
x_{13}	2.5	TRUE
x_{10}	2.2	TRUE
x_8	1.5	TRUE
x_4	0.5	FALSE
x_9	0	FALSE
x_{11}	0	FALSE

(b) Variables

Level	Assignments
1	x_1, x_2
2	x_3, $\neg x_4$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$
5	x_7, x_8, x_9, x_{10}

(c) Assignments

Figure 5.1: The status before backtracking

$\neg x_7 \vee \neg x_2$ is learnt by 1-UIP [49] and thus $dl_{asrt} = 1$. According to VSIDS, the solver will only increase the activity scores (assuming the increment is 1) of the variables involving in the conflict, namely, $\{x_2, x_7, x_8, x_9, x_{10}\}$. Therefore, the activity scores of the variables assigned between dl_{conf} and dl_{asrt}, $\{x_3, x_4, x_5, x_6, x_{12}, x_{13}\}$, remain the same. As shown in Figure 5.2c, in the decision immediately after backtracking to dl_{asrt}, x_3 will be chosen and assigned TRUE again at the level 2. Note that the resulting set of variable assignments at the level 2 is a superset of that before backtracking. The set of variable assignments at the level 3 is also similar to that before backtracking, except that x_6 has been "lifted" to the level 2.

By comparing the variable assignments before and after each backtracking, we have Figure 5.3 that shows the percentage of discarded variable assignments that are chosen as decisions or propagated again before the next backtracking. It is interesting to see that the solver tends to either enter a totally different search space or stubbornly stick to its previous choices. But

$\neg x_1 \vee x_2$
$\neg x_3 \vee \neg x_4$
$\neg x_1 \vee x_4 \vee x_5 \vee x_6$
$x_5 \vee x_{13}$
$\neg x_7 \vee x_8$
$\neg x_7 \vee x_9$
$\neg x_2 \vee \neg x_8 \vee x_{10}$
$\neg x_8 \vee \neg x_9 \vee \neg x_{10}$
$x_4 \vee x_7 \vee \neg x_{11}$
$x_7 \vee x_{11} \vee x_{12}$
$x_6 \vee x_{11}$
$\neg \mathbf{x_7} \vee \neg \mathbf{x_2}$

(a) Clauses

Variable	Activity Score	Last Values
x_1	10	TRUE
x_2	**8.2**	TRUE
x_3	8.1	TRUE
x_7	**6.5**	TRUE
x_5	6.4	FALSE
x_{12}	6	FALSE
x_6	3.7	**TRUE**
x_{10}	**3.2**	TRUE
x_8	**2.5**	TRUE
x_{13}	2.5	TRUE
x_9	**1**	**TRUE**
x_4	0.5	FALSE
x_{11}	0	FALSE

(b) Variables

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}, x_6
3	$\neg x_5$, x_{13}
4	x_{10}
5	x_8, $\neg x_9$

(c) Assignments

Figure 5.2: The status after backtracking

for a majority of backtrackings, a large proportion of discarded variable assignments are repeated. Note that we only consider those backtrackings that go back more than 10 levels and do not take account of restarts. Also, the variable assignments on the conflicting level are not counted in computing this percentage.

5.3 Partial Backtracking

In this section, we present the partial backtracking strategy that allows the solver to backtrack to some level dl_{back} such that $dl_{conf} > dl_{back} \geq dl_{asrt}$, therefore saving the propagations between dl_{back} and dl_{asrt}.

There are two reasons that classic backtracking prefers to use the assertion level as the backtracking level. First, after each backtracking, the learnt clause becomes unit and hence

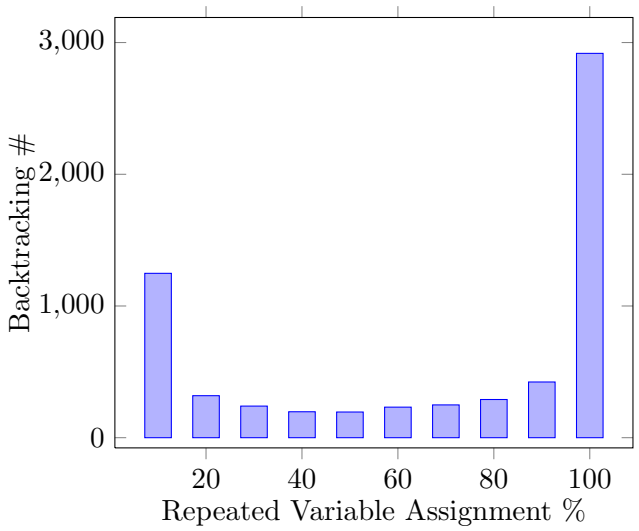


Figure 5.3: Repeated variable assignment percentage while solving ACG-15-5p1.cnf from SAT Challenge 2012

BCP can be invoked. Second, the succeeding BCP will not cause any consistency issue. To adopt the partial backtracking strategy, we need to update BCP procedure so that the two conditions are still met.

The first condition can be easily satisfied by backtracking to any level lower than dl_{conf} but higher than or equal to dl_{asrt}. We note that the assertion level is the lowest level that the solver can backtrack to while keeping the learnt clause unit. The main complications come from maintaining the second condition. There are four kinds of issues BCP may encounter after backtracking to a level higher than dl_{asrt}. In Section 5.3.1, we will discuss these issues and give the corresponding solutions at clause level. The complete solution will be given in Section 5.3.2.

5.3.1 Complications and Solutions for Partial Backtracking

Unusual Implication. Classic backtracking guarantees that the solver always obtains implications at the current level dl_{curr}, that is, for any implication $l@dl$ in the implication queue, $dl = dl_{curr}$ (see Algorithm 2). However, this is not true for partial backtracking. A simple counterexample is the implication obtained from the learnt clause. This implication is at dl_{asrt}, which is lower than or equal to dl_{curr} after backtracking partially ($dl_{curr} = dl_{back} \geq dl_{asrt}$).

Moreover, this implication may result in more implications, which can be scattered at any level between dl_{asrt} and dl_{curr}.

To the best of our knowledge, no existing solver exploits this guarantee in any essential way. In the implementation of Nigma, we simply relax this restriction.

Inappropriate Watched Literal. Generally, if a clause becomes unit and its sole free literal gets assigned according to this implication, its watched literals are certainly assigned at the highest decision level among all its literals. This condition may be violated after backtracking partially.

Consider a clause $x_1 \vee \neg x_2 \vee x_3$. Suppose x_3 is assigned FALSE at the level 10, and x_1 and x_2 are free. So x_1 and $\neg x_2$ are watched for this clause. During BCP after backtracking partially, x_1 may be assigned FALSE at the level 6. In this case, it is inappropriate to still watch x_1. Since the level of x_3 is higher than the level of x_1, x_3 should be watched instead.

In order to solve this issue, we use the following procedure, where $\delta(l)$ is a function that returns the decision level where the literal l gets assigned.

- $AdjustWatchedLiteral(wl, c)$

  Pre-condition: The literal wl is watched in the clause c; All the unwatched literals in c are FALSE.

  Description: Search for an unwatched literal l in c such that $\delta(l) > \delta(wl)$ and for any unwatched literal l' in c, $\delta(l) \geq \delta(l')$. If successful, unwatch wl, watch l and return l. Otherwise, return wl.

Spurious Conflict. As we noted before, BCP may lead to conflicts. A standard conflict has the following implicit feature: the two FALSE literals with the highest levels in the conflicting clause are assigned at the same level. However, during BCP after backtracking partially, the solver might encounter a *spurious conflict* where these two literals are assigned at different levels.

We give a simple example to illustrate the spurious conflict. Consider a clause $x_1 \vee \neg x_2$. After backtracking partially, we may have two implications $\neg x_1$@10 and x_2@15 at the same

time. This is a conflict (as all the literals are FALSE), but it is different from the standard one.

The spurious conflict cannot be resolved by the standard learning procedure. From another perspective, the spurious conflict essentially implies that the FALSE literal with the highest level should have been implied at the second highest level among the literals in the conflicting clause. In other words, without learning, we can immediately obtain an implication by simply backtracking to a level between the highest level and the second highest level in the conflicting clause. That level can also be but not necessary the second highest level because we are able to handle the unusual implication now. We have the following procedure to resolve spurious conflicts.

- $ResolveSpuriousConflict(c)$

  Pre-condition: All the literals in the clause c are FALSE; The literals wl_1 and wl_2 are watched in c; $\delta(wl_1) \neq \delta(wl_2)$.

  Description: If $\delta(wl_1) > \delta(wl_2)$, backtrack to the level $\delta(wl_1)-1$ and push the implication $wl_1@\delta(wl_2)$ into the implication queue. If $\delta(wl_1) < \delta(wl_2)$, backtrack to the level $\delta(wl_2)-1$ and push the implication $wl_2@\delta(wl_1)$ into the implication queue.

Wrong Decision Level. After backtracking partially, some assigned variables need to update their decision levels. For example, consider a clause $x_1 \vee x_2$. Initially, x_1 is assigned TRUE at the level 18 and x_2 is free. Suppose at the level 20, a conflict is identified and the solver backtracks to the level 19 while $dl_{asrt} = 5$. Further suppose that the succeeding BCP induces the implication $\neg x_2@15$. As a result, the decision level of x_1 should be modified to 15. The issue can be solved by backtracking to the level 17 and get the implication $x_1@15$. The following procedure is used for this purpose.

- $ResolveWrongDecisionLevel(c)$

  Pre-condition: All the unwatched literals in the clause c are FALSE; c has a TRUE watched literal wl_{true} and a FALSE watched literal wl_{false}; $\delta(wl_{true}) > \delta(wl_{false})$.

  Description: Backtrack to the level $\delta(wl_{true})-1$ and push the implication $wl_{true}@\delta(wl_{false})$ into the implication queue.

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$

(a)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$
3	$\neg x_5$, x_6, x_{13}
4	$\neg x_{12}$

(b)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}
3	$\neg x_5$, x_6, x_{13}
4	

(c)

Level	Assignments
1	x_1, x_2, $\neg x_7$
2	x_3, $\neg x_4$, $\neg x_{11}$, x_{12}, x_6
3	
4	

(d)

Figure 5.4: The status after backtracking partially

Both processes of resolving spurious conflict and wrong decision level might trigger further backtracking. A helper procedure, *ClearInvalidImplications*, is defined to adjust the implication queue accordingly.

- *ClearInvalidImplications*()

  Description: Remove invalid implications from the implication queue. An implication $l@dl$ is *invalid* if $dl > dl_{curr}$.

In spite of the possible chained backtracking, whenever BCP terminates, the current decision level is always higher than or equal to the assertion level.

5.3.2 BCP after Partial Backtracking

As mentioned before, the standard BCP needs an adjustment if the solver takes a partial backtracking. Algorithm 4 shows the procedure *PropagateAmending* that is a special propagating procedure to be used after backtracking partially. Algorithm 5 shows the procedure *BCPAmending* that replaces the standard BCP procedure.

Let us revisit the example in Section 5.2. At this time, when the conflict occurs at the level 5, the solver takes a partial backtracking to the level 4 (see Figure 5.4a). While propagating the implication $\neg x_7@1$, the solver obtains $\neg x_{11}@2$ (unusual implication) (see Figure 5.4b) due

Algorithm 4 *PropagateAmending*($l@dl$)

```
1: wl_1 ← ¬l
2: for all clause c where wl_1 is watched do
3:    Search for a non-FALSE unwatched literal l' in c
4:    if Exists l' then
5:       Unwatch wl_1
6:       Watch l'
7:    else
8:       wl_1 ← AdjustWatchedLiteral(wl_1, c)
9:       wl_2 ← the other watched literal in c
10:      if wl_2 is FALSE then
11:         if δ(wl_1) > δ(wl_2) then
12:            wl_2 ← AdjustWatchedLiteral(wl_2, c)
13:         end if
14:         if δ(wl_1) == δ(wl_2) then
15:            Backtrack to δ(wl_1)
16:            ConflictAnalysis() {Standard conflict}
17:            ClearInvalidImplications()
18:            return
19:         else
20:            ResolveSpuriousConflict(c) {Spurious conflict}
21:            ClearInvalidImplications()
22:         end if
23:      else if wl_2 is TRUE then
24:         if δ(wl_2) > δ(wl_1) then
25:            ResolveWrongDecisionLevel(c) {Wrong decision level}
26:            ClearInvalidImplications()
27:         end if
28:      else
29:         ImplicationQueue.Push(wl_2@δ(wl_1))
30:      end if
31:   end if
32: end for
```

Algorithm 5 *BCPAmending*()

```
1: while ImplicationQueue is not empty do
2:    l@dl ← ImplicationQueue.pop()
3:    PropagateAmending(l@dl)
4: end while
```

to $x_4 \vee x_7 \vee \neg x_{11}$. In the next iteration of propagation, the solver identifies a spurious conflict ($x_7 \vee x_{11} \vee x_{12}$) and has to go back one level to resolve it (see Figure 5.4c). Due to the existence of $x_6 \vee x_{11}$, x_6 should have been implied at the level 2 (wrong decision level), so the solver goes back one level again (see Figure 5.4d). Then BCP terminates because no more implication or conflict can be found. It is clearly seen that the solver amends the existing assignment trail conservatively, not simply discarding a significant portion of it. We note that under this strategy, it is possible that the solver enters a search space which is quite different from the one resulting from the classic backtracking.

We shall point out that, when the implication to be propagated happens to be at the current level, the effect of *PropagateAmending* is exactly the same as *Propagate*. This indicates that *PropagateAmending* is essentially a generalization of *Propagate*.

5.4 Optimization

In this section, we discuss optimizations applicable to the algorithms *PropagateAmending* and *BCPAmending*.

First, the implication queue can be constructed as a priority queue. As we described before, most CDCL solvers organize implications in a queue and propagates them in FIFO manner. However, since the implications in the queue can be scattered on different levels, unnecessary propagations can be avoided by giving higher priority to the implication at the lowest level in the queue. The intuition is that propagation may induce backtracking due to spurious conflict and wrong decision level, making some implications invalid and removed from the queue. For example, suppose that we have the implications x_1@10 and $\neg x_2$@20 in the implication queue. If propagating x_1@10 incurs a backtracking to some level lower than 20, $\neg x_2$@20 becomes invalid and the solver needs not propagate it.

Second, even if encountering a standard conflict in *PropagateAmending*, it is possible to postpone the conflict analysis. Suppose, while propagating x_1@10, the solver meets a standard conflict at the level 20. If the solver does not analyse the conflict immediately but continues propagating, it may backtrack to some level lower than 20 later due to spurious conflict or wrong decision level, making that conflict disappear automatically.

Third, it is unnecessary to call *PropagateAmending* in each iteration of *BCPAmending*. As mentioned before, *PropagateAmending* is a generalization of *Propagate* and it is more expensive than *Propagate*. If the implication to be propagated happens to be at the current level, calling *Propagate* directly instead of *PropagateAmending* will not cause any issue.

Fourth, it is also unnecessary to backtrack partially every time a conflict occurs. The motivation of partial backtracking is to save propagations. Thus this strategy should be more efficient if a large number of propagations are going to be discarded or repeated. In Nigma, we measure the saving by the number of levels the solver would go back by classic backtracking, namely, $dl_{conf} - dl_{asrt}$. According to our experiments, when we set the triggering condition to $dl_{conf} - dl_{asrt} > 10$, around $5\% \sim 30\%$ of conflicts will trigger partial backtracking.

5.5 Experiment Results

In this section, we present experiment results using our solver Nigma, which is a CDCL solver based on MiniSat 2.2 [14]. The benchmark suite consists of the 600 instances from the application track of SAT Challenge 2012 [4]. We conducted experiments on a 3.40GHz $\times$ 8 Intel Core i7-2600K processor with 900 second timeout and 7GB memory limit per instance.

The versions of Nigma with partial backtracking and with classic backtracking are denoted by Nigma-PB and Nigma-CB, respectively. Nigma-PB is configured as follows: if $dl_{conf} - dl_{asrt} \leq 10$, the solver simply follows the classic backtracking strategy; otherwise, the solver backtracks only one level, that is, it backtracks to the level $dl_{conf} - 1$. We use Glucose 2.2 [1] as an additional reference.

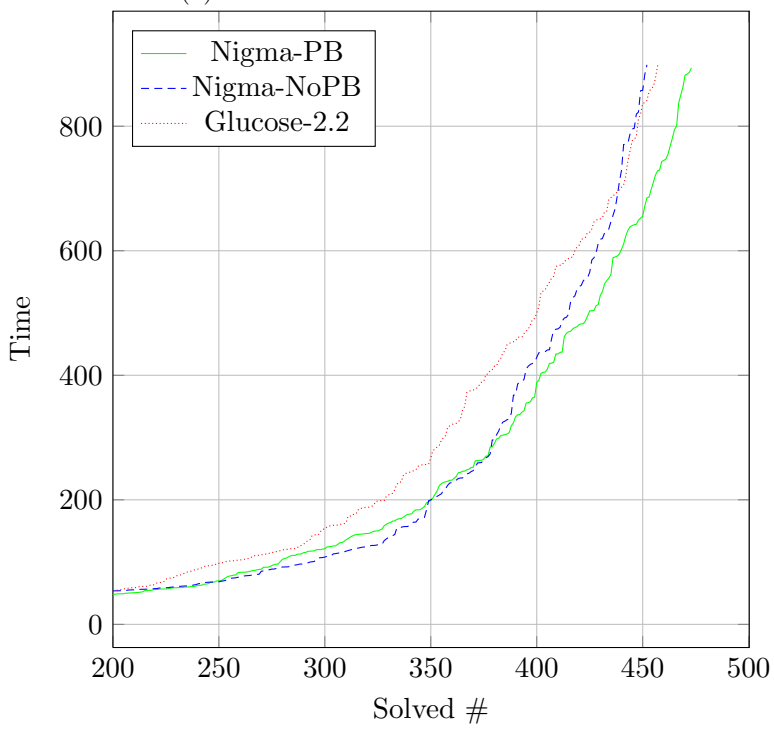
Figure 5.5a shows the number of instances solved by the three solvers and Figure 5.5b is the cactus plot of the results. It is clearly seen that when applying partial backtracking, Nigma-PB solved 21 more instances than Nigma-CB, and it also performs better than Glucose 2.2.

An in-depth view of the effect of partial backtracking is given in Figure 5.6, showing the percentage of fewer levels the solver backtracks for each solved instance. We note that, for a majority of instances, when the solver takes a partial backtracking, it backtracks $10\% \sim 60\%$ fewer levels finally, compared with classic backtracking.

We also compare two additional metrics in the experiment, in order to explain the perfor-

Solver	SAT	UNSAT	Solved #
Nigma-PB	222	251	473
Nigma-CB	212	240	452
Glucose-2.2	212	246	458

(a) The Number of Solved Instances



(b) Runtime Cactus Plot

Figure 5.5: Experiment results of Nigma-PB, Nigma-CB and Glucose 2.2 on the benchmark suite from the application track of SAT Challenge 2012

mance improvement by partial backtracking from a different perspective. The first metric is the number of decisions to solve an instance. Generally speaking, fewer decisions indicate the solver explores the search space in a better way [49]. According to the experiment, among the 439 instances solved by both Nigma-PB and Nigma-CB, 317 instances are solved by Nigma-PB with fewer decisions than by Nigma-CB.

The second metric is the number of decisions per conflict for a solved instance. We are interested in this metric because the power of CDCL solvers stems from identifying and learning from conflicts. The number of decisions per conflict reflects how frequently the solver identifies a conflict. The smaller this number is, the more often the solver detects and corrects its fault

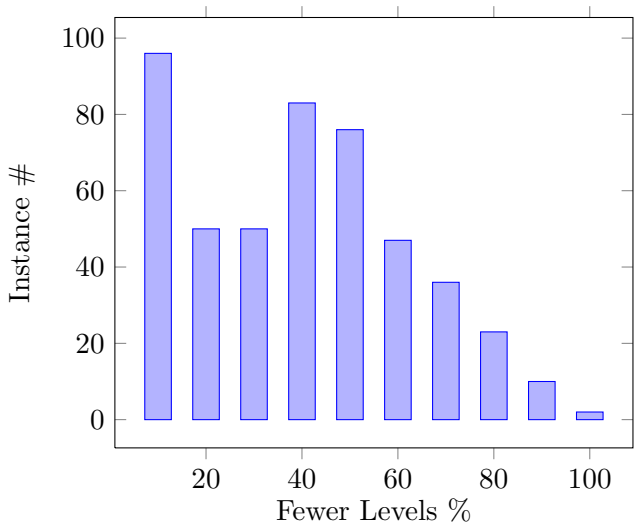


Figure 5.6: Nigma backtracks fewer levels with partial backtracking

in making decisions. Partial backtracking has the potential to reduce this number as the solver might detect a standard conflict at a level higher than dl_{asrt} (see Line 14-18 in Algorithm 4) while retaining the ability to detect a standard conflict at dl_{asrt}. The experiment result confirms our conjecture: 387 instances are solved by Nigma-PB with fewer decisions per conflict than by Nigma-CB.

5.6 Summary

In this chapter, we presented the partial backtracking strategy which is essentially an extension of classic backtracking. This strategy amends the assignment trail instead of simply discarding a portion of it. As a result, some propagations need not to be repeated and the solver can go deeper in certain search space. Our experiments show that this new kind of backtracking improves the performance of CDCL solvers. Besides the optimizations mentioned in Section 5.4, we are investigating the following two aspects to further improve its efficiency.

First, in our current implementation, the solver backtracks to $dl_{asrt} - 1$ first. In fact, any level higher than dl_{asrt} can be used for the initial backtracking, as going back to that level still keeps the learnt clause unit. We are interested in designing a better heuristic to select the initial backtracking level.

Second, we would explore other criteria to trigger a partial backtracking. A promising candidate is the number of variable assignments the solver would discard by taking a classic backtracking.

CHAPTER 6. SUMMARY AND DISCUSSION

In this thesis, we surveyed various algorithms and implementations in building efficient SAT solvers. We also introduced our contribution, partial backtracking, which is an extension of the classic backtracking strategy. With partial backtracking, the solver consecutively amends the variable assignments instead of discarding them completely so that it does not backtrack as many levels as classic backtracking does after analyzing a conflict. This new strategy has been implemented in out solver Nigma and the experiment results show that Nigma benefits from this adjustment.

The organization of SAT solvers remains stable for decades and it seems hard to improve them from a structural perspective. Future research on SAT solvers mainly focuses on discovering more powerful heuristics. The performance of a SAT solver may vary greatly even if a small change in the heuristics used by its components.

Besides, as the problems from a specific domain may have special structures, it is a good direction to develop domain-specific SAT solvers or techniques [43][42], even though the SAT solver was designed as a general-purpose tool initially.

BIBLIOGRAPHY

[1] Gilles Audemard and Laurent Simon. Glucose. https://www.lri.fr/~simon/?page=glucose.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st international joint conference on Artificial Intelligence*, pages 399–404, 2009.

[3] Gilles Audemard and Laurent Simon. Refining Restarts Strategies for SAT and UNSAT. In *Proceedings of the 18th international conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer-Verlag, 2012.

[4] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Sat challenge 2012. http://baldur.iti.kit.edu/SAT-Challenge-2012/index.html.

[5] Armin Biere. Lingeling. http://fmv.jku.at/lingeling/.

[6] Armin Biere. Adaptive Restart Control for Conflict Driven SAT Solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, pages 28–33. Springer-Verlag, 2008.

[7] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, (75-97):45, 2008.

[8] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in computers*, 58:117–148, 2003.

[9] Geoffrey Chu, A. Harwood, and P.J. Stuckey. Cache conscious data structures for boolean

satisfiability solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:99–120, 2009.

[10] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal methods in system design*, 19(1):7–34, 2001.

[11] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[12] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[13] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 8th international conference on Theory and Applications of Satisfiability Testing*, pages 61–75. Springer-Verlag, 2005.

[14] Niklas Eén and Niklas Sörensson. Minisat. http://minisat.se/.

[15] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 333–336. Springer, 2004.

[16] Evgueni Goldberg and Yakov Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exhibition*, pages 142–149, Paris, France, 2002.

[17] Carla P Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*, pages 121–135. Springer Berlin Heidelberg, 1997.

[18] Carla P Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning*, 24:67–100, 2000.

[19] Carla P Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the fifteenth national/tenth conference on Artificial in-*

telligence/Innovative applications of artificial intelligence, pages 431–437, Madison, Wisconsin, USA, 1998. American Association for Artificial Intelligence.

[20] Marijn Heule. March. http://www.st.ewi.tudelft.nl/~marijn/.

[21] Steffen Hölldobler, Norbert Manthey, and Ari Saptawijaya. Improving resource-unaware SAT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, number Cdcl, pages 519–534, 2010.

[22] Andrei Horbach, Thomas Bartsch, and Dirk Briskorn. Using a SAT-solver to schedule sports leagues. *Journal of Scheduling*, 15(1):117–125, 2012.

[23] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th international joint conference on Artificial intelligence*, pages 2318–2323. Morgan Kaufmann Publishers Inc., 2007.

[24] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis*, volume 25, pages 14–25. ACM, 2000.

[25] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 129–144. Springer-Verlag, 2010.

[26] Chuan Jiang and Ting Zhang. Partial backtracking in cdcl solvers. In *Proceedings of the 19th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning*, pages 490–502, Stellenbosch, South Africa, 2013. Springer-Verlag.

[27] Henry A Kautz and Bart Selman. Planning as Satisfiability. In *European Conference on Artificial Intelligence*, volume 92, pages 359–363, 1992.

[28] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(89):173–180, 1993.

[29] Inês Lynce, Luis Baptista, and João P. Marques-Silva. Stochastic systematic search algorithms for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:190–204, 2001.

[30] Inês Lynce and João P. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, pages 105–110. IEEE Computer Society, 2003.

[31] Inês Lynce and João P. Marques-Silva. Random backtracking in backtrack search algorithms for satisfiability. *Discrete Applied Mathematics*, 155(12):1604–1612, 2007.

[32] Norbert Manthey. Riss. http://www.ki.inf.tu-dresden.de/~norbert/html/riss.php.

[33] Norbert Manthey and Ari Saptawijaya. Towards improving the resource usage of SAT-solvers. In *Pragmatics of SAT Workshop*, 2010.

[34] João P. Marques-Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[35] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th conference on Design Automation*, pages 530–535, New York, USA, 2001.

[36] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulae. In *Proceedings of the 18th European Conference on Artificial Intelligence*, pages 525–529, 2008.

[37] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proceedings of the 10th international conference on Theory and Applications of Satisfiability Testing*, pages 294–299. Springer-Verlag, 2007.

[38] Lawrence Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[39] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In *Proceedings of the 2nd DIMACS Challange on Cliques, Coloring, and Satisfiability*, pages 521–532, 1993.

[40] Bart Selman, Hector Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th national conference on Artificial intelligence*, pages 440–446, San Jose, California, USA, 1992. AAAI Press.

[41] Mate Soos. Cryptominisat. http://www.msoos.org/cryptominisat2/.

[42] Mate Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In *Pragmatics of SAT*, Edinburgh, Scotland, 2010.

[43] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 244–257, 2009.

[44] Niklas Sörensson and Armin Biere. Minimizing Learned Clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243. Springer, 2009.

[45] Paul Stephan, Robert K Brayton, and Alberto L Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(9):1167–1176, 1996.

[46] Sathiamoorthy Subbarayan and Dhiraj K Pradhan. NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, pages 276–291. Springer-Verlag, 2005.

[47] Peter van der Tak, Antonio Ramos, and Marijn Heule. Reusing the assignment trail in cdcl solvers. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:133–138, 2011.

[48] Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT based scheduler for tournament schedules. In *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing*, pages 10–13, 2004.

[49] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of the 2001*

IEEE/ACM international conference on Computer-aided design, pages 279–285. IEEE Press, 2001.