# EFFICIENT ALGORITHMS FOR CLAUSE-LEARNING SAT SOLVERS

by

Lawrence Ryan

B.Sc., Simon Fraser University, 2002

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
M.Sc.
in the School
of
Computing Science


SIMON FRASER UNIVERSITY
February 2004

## APPROVAL

**Name:** Lawrence Ryan

**Degree:** M.Sc.

**Title of thesis:** Efficient Algorithms for Clause-Learning SAT Solvers

**Examining Committee:** Dr. Pavol Hell
Chair

---

Dr. David G. Mitchell, Senior Supervisor

---

Dr. Lou Hafer, Supervisor

---

Dr. Delgrande, SFU Examiner

**Date Approved:** _____

# Abstract

Boolean satisfiability (SAT) is NP-complete. No known algorithm for SAT is of polynomial time complexity. Yet, many of the SAT instances generated as a means of solving real-world electronic design automation problems are simple enough, structurally, that modern solvers can decide them efficiently. Consequently, SAT solvers are widely used in industry for logic verification. The most robust solver algorithms are poorly understood and only vaguely described in the literature of the field. We refine these algorithms, and present them clearly. We introduce several new techniques for Boolean constraint propagation that substantially improve solver efficiency. We explain why literal count decision strategies succeed, and on that basis, we introduce a new decision strategy that outperforms the state of the art. The culmination of this work is the most powerful SAT solver publically available.

# Contents

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview

Boolean satisfiability (SAT) is NP-complete. If $P$ is a proper subset of $NP$, there exists no algorithm $g$ such that for some polynomial function, $f$, $g$ decides any given SAT instance in time bounded above by $f$ applied to the instance's length.

By sufficiently restricting an intractable problem, one can always obtain a tractable problem. Some SAT instance classes are solvable in polynomial time. For example, instances containing no clause with more than two different literals can be solved in linear time[9]. Instances containing no clause with more than one positive literal are solvable in linear time[9]. Algorithms that take exponential time for some classes may take polynomial time for others. We are interested in algorithms with the capacity to

- solve all SAT instances; and
- solve efficiently many instances generated through the application of SAT to practical problems arising in contemporary industry.

This work is concerned primarily with solving SAT instances that have been generated as a means of solving real-world electronic design automation problems. It is empirically evident that many of these instances are characterized by structural properties permitting efficient solution[7]. But, it is also the case that many interesting formulas are challenging or even impossible in reasonable time for current solvers. Through careful implementation and

the use of powerful new search techniques we produce a solver that dramatically outperforms state of the art SAT solvers such as zChaff[30] and berkmin56[16].

Boolean satisfiability is well-studied, and SAT solvers have a long history. The Davis-Putnam (DP) algorithm[12], published in 1960, is typically cited as the first entry in the solver timeline. But, as noted in [8], an essentially identical algorithm was in fact developed half a century earlier, by L. Löwenheim[17]. DP is very inefficient at finding a satisfying assignment when one exists. This fact, and the fact that DP's memory consumption is explosive in practice, motivated the subsequent development of the Davis-Logemann-Loveland (DLL) algorithm[11].

Over the past decade, several powerful DLL solvers have been introduced, including POSIT[14], Satz[25], and most recently, kcnfs[13]. These solvers deviate from the original DLL algorithm primarily in that they use better decision heuristics. Such solvers are still state of the art for proving random instances unsatisfiable. (Incomplete local-search programs, such as unitwalk[18], are more successful on satisfiable random instances.)

Conflict-driven clause learning was first used in the solvers GRASP[29] and Relsat[3]. By appending implicates to the formula, and by using these implicates to achieve non-chronological backtracking, these solvers far outperform the best DLL solvers on non-random instances. These techniques are the basis of all the most successful modern solvers for structured industrial instances.

The Chaff solver made GRASP and Relsat obsolete. Chaff's most important innovation was a decision strategy that allowed it to take better advantage of conflict-driven clause learning. Chaff also introduced a variation on the two-pointer BCP algorithm published several years earlier as part of the SATO solver[40]. Because Chaff was designed to be applied in an industrial setting, heavy emphasis was placed on efficient implementation.

Our work extends along the lines laid out by Chaff. We have developed a SAT solver that is much faster than the best of the Chaff variants on many interesting problem classes. This progress is guided by our new explanations for the power of modern solver algorithms.

In Chapter 2, we present basic learning and non-chronological backtracking algorithms. We also propose and support an explanation for the remarkable power of conflict-driven clause learning. In Chapter 3, we explain why two-pointer BCP algorithms are fast. We then introduce some straightforward, but very effective, improvements to the state of the art. Finally, in Chapter 4, we provide the first plausible explanation for why Chaff's decision strategy succeeds. This explanation accurately predicts the superiority of an elegant new

decision strategy, which we describe in detail.

## 1.2 Experimental Data

### 1.2.1 Standard Benchmarks

In Table 1.1, we present solver runtimes for several well-known benchmarks. Our siege SAT solver implements our powerful new VMTF decision strategy and the BCP techniques described in Chapter 3. Results are listed for version 1 of siege. Version 1 is a slight variation on version 0, the program that competed in the SAT2003 competition. Berkmin561 is not evaluated here, because (a) it uses an unpublished decision strategy, and (b) for legal reasons, the company that owns berkmin has stopped distributing executables to the public at this time.

Note that berkmin's performance on the hanoi planning benchmarks differs from what is reported in [16]. This is a result of the SAT2002 instance shuffling. Berkmin is "lucky" on the original instance, but randomization reveals that it is not actually able to solve hanoi6 fast.

| **suite** | **#instances** | **zChaff** | **berkmin56** | **siege_v1** |
|---:|:---:|:---:|:---:|:---:|
| fvp-unsat.1.0[37] | 4 | 1,032 | 953 | 138 |
| vliw-sat1.0[37] | 100 | 7,956 | 5,307 | 434 |
| fvp-unsat.2.0[37] | 22 | >72,000 | 4,861 | 3,046 |
| hanoi4,5,6[34] | 3 | >72,000 | 24,346 | 1,125 |
| IBM_FV_2002_01_rule[38] | 20 | 43,986 | >72,000 | 47,583 |
| miters[28] | 25 | 775 | 344 | 197 |

Table 1.1: Runtime in seconds on a SunBlade1000 750MHz Ultra I

### 1.2.2 The SAT2003 Competition

Version 0 of siege participated in the first round of the SAT2003 competition. It performed well in both of the non-random benchmark categories. But, because it was entered late, siege was barred from participating in the second round.

Competition benchmark instances are clustered into series. A solver is said to have solved a series if it has solved at least one instance in the series. Every solver was given 15 minutes to decide each instance on an 1800 MHz Athlon with 1 gigabyte of RAM. The

| solver | series | instances | solver | series | instances |
|---|---|---|---|---|---|
| forklift | 34 | 143 | jerusat1c | 27 | 119 |
| berkmin561 | 32 | 136 | sato | 25 | 101 |
| **siege_v0** | **32** | **135** | opensat | 24 | 94 |
| berkmin62 | 32 | 133 | satzilla | 18 | 83 |
| bmsat | 31 | 132 | satzilla2 | 18 | 82 |
| satzoo1 | 31 | 124 | marchtt | 18 | 67 |
| satzoo0 | 31 | 123 | marchsp | 17 | 63 |
| oepir | 30 | 126 | xqingting | 9 | 44 |
| funex | 29 | 127 | lsat | 8 | 46 |
| satnik | 29 | 116 | farseer | 8 | 28 |
| jquest2 | 29 | 116 | tts | 7 | 20 |
| zchaff | 29 | 115 | kcnfs | 6 | 17 |
| jerusat1b | 28 | 120 | qingting | 4 | 17 |
| jerusat1a | 28 | 119 | unitwalk | 4 | 15 |
| limmat | 28 | 114 | saturn | 3 | 22 |

Table 1.2: Results for industrial benchmarks

solvers were ranked, first by the number of series solved, then by the number of benchmarks solved. The reader is referred to [33] for further details.

We list the first round results for the non-random benchmark categories in Tables 1.2 and 1.3. In the industrial category, siege ranked third, ahead of berkmin62 (an unpublished solver, superior to berkmin56). Siege was outperformed by forklift (an improvement on the unreleased berkmin621) and berkmin561. zChaff ranked $12^{th}$.

In the handmade category, siege took first place. Note that forklift and the other berkmin solvers did poorly, worse even than zChaff. The difference between the handmade and industrial categories is that the former includes instances that do not encode electronic design automation problems.

## 1.3 Basics

A *literal* is a signed Boolean variable. If $x$ is a Boolean variable, we use $x$ to denote its non-negated literal, and $\overline{x}$ to denote its negated literal. If the variable $x$ is $true$, the literal $x$ is $true$ and the literal $\overline{x}$ is $false$. If the variable $x$ is $false$, the literal $x$ is $false$ and the literal $\overline{x}$ is $true$. If the variable $x$ has not been assigned a truth value, it is described as *free*. This term extends to (the necessarily unvalued) literals of a free variable. The

| solver | series | instances | solver | series | instances |
|---|---|---|---|---|---|
| **siege_v0** | **22** | **97** | berkmin62 | 16 | 60 |
| jerusat1b | 22 | 81 | jquest2 | 16 | 57 |
| satzoo1 | 21 | 102 | kcnfs | 16 | 51 |
| satzilla2 | 21 | 90 | lsat | 15 | 128 |
| satzilla | 21 | 89 | bmsat | 15 | 53 |
| satnik | 20 | 88 | funex | 15 | 53 |
| jerusat1a | 20 | 82 | sato | 15 | 52 |
| marchsp | 19 | 84 | limmat | 14 | 52 |
| jerusat1c | 19 | 63 | tts | 10 | 68 |
| marchtt | 18 | 83 | xqingting | 10 | 38 |
| satzoo0 | 18 | 82 | opensat | 8 | 25 |
| zchaff | 17 | 67 | saturn | 7 | 29 |
| oepir | 17 | 64 | farseer | 6 | 14 |
| forklift | 17 | 63 | unitwalk | 5 | 20 |
| berkmin561 | 16 | 65 | qingting | 3 | 12 |

Table 1.3: Results for handmade benchmarks

negation of the literal $x$ is $\overline{x}$, and vice versa. A literal takes on a value only with respect to an assignment of truth values to variables. If it is clear from context, we do not explicitly cite the causative assignment when describing a literal as $true$, $false$, or free.

A *clause* represents a disjunction of literals. For simplicity, henceforth the logical relationship between elements of a clause is left implicit; clauses will be treated as sets of literals. Where the contents of a clause are specified, the following notation is used. Square brackets are used for delimitation. Between them, individual literals are denoted as described above, and uppercase letters stand for sets of literals. For example, $[a]$ denotes a clause containing only the literal $a$. $[a\overline{b}]$ denotes a clause containing the literals $a$ and $\overline{b}$. $[a\overline{b}R]$ denotes a clause that contains $a$, $\overline{b}$, and in addition, the (perhaps empty) set of literals, $R$. The term *unit clause* refers to a clause with exactly one literal. If a clause $c$ contains one or more literals that evaluate to $true$ under truth assignment $t$, we say $c$ is satisfied under $t$.

An instance of the SAT problem consists of a conjunction of clauses, $F$. The term *CNF* is used to refer to such formulas. Normally we will treat a CNF as if it were simply a set of clauses. Let $V$ be the set of those variables that have a literal in formula $F$. If there exists an assignment of truth values to members of $V$ under which every clause in $F$ is satisfied, $F$ is satisfiable. Otherwise, $F$ is unsatisfiable.

## 1.4 The DP and DLL Algorithms

Most competitive modern SAT solvers are, essentially, derivatives of the Davis-Logemann-Loveland (DLL) algorithm[11]. The DLL algorithm is a variation on the Davis-Putnam (DP) algorithm[12]. The following exposition sets out a foundation for later refinement.

### 1.4.1 The Davis-Putnam Algorithm

The input to the algorithm is a CNF, $F$.

**PR.** In case the input contains an empty clause, return "unsatisfiable".

**P0.** Remove from $F$ all clauses that contain at once both literals $x$ and $\overline{x}$ of some variable $x$. If $F$ is now empty, return "satisfiable".

**P1.** If there is a variable $x$ such that $F$ contains both $[x]$ and $[\overline{x}]$, return "unsatisfiable".

**P2.** If there is a literal $x$ for which $F$ contains $[x]$, delete from $F$ all clauses containing $x$, and remove the negation of $x$ from all clauses in which it is present. If $F$ is now empty, return "satisfiable". If $F$ contains a unit clause, go back to step P1.

**P3.** While there is a pure literal in $F$, delete all clauses in which a pure literal is present. If $F$ is now empty, return "satisfiable".

**P4.** Select some variable, $x$, that has a literal in one of the shortest clauses in $F$. Let $A$ be the conjunction of all clauses from $F$ that contain the literal $x$. Let $B$ be the conjunction of all clauses from $F$ that contain the literal $\overline{x}$. Let $C$ be the conjunction of all clauses that contain neither $x$ nor $\overline{x}$. Shorten every clause in $A$ by removing $x$, to produce $A'$. Shorten every clause in $B$ by removing $\overline{x}$, to produce $B'$. Replace $F$ with $(A' \vee B') \wedge C$. Use distribution to transform $F$ into a conjunction of clauses. Go back to step P0.

Step P0 removes clauses that are trivially satisfied. If all clauses in $F$ are of this type, $F$ is itself trivially satisfiable.

Steps P1 and P2 constitute *boolean constraint propagation* (BCP). It is the need to perform BCP efficiently that drives much of the work reflected in competitive modern SAT solvers. In step P1, the formula is checked for contradictory unit clauses. The presence of

$[x]$ implies that $x$ must be true; the presence of $[\overline{x}]$ implies that $x$ must be false. If both are present the formula is unsatisfiable.

In step P2, the occurence of a literal $x$ in a unit clause is used to justify simplification of $F$. If $x$ is $false$ then the unit clause $[x]$ is unsatisfiable. Thus $x$ being $true$ is a necessary consequence of any truth assignment that satisfies all clauses in $F$. All instances of $x$ are removed through *unit-subsumption*: if a clause has as a subset the satisfied clause $[x]$, it must itself be satisfied and so can be removed from further consideration. All instances of the negation of $x$ are eliminated through *unit-resolution*. Since the negation of $x$ is $false$, it cannot contribute to satisfying a clause in which it occurs. Therefore it may be excised from such clauses—they will necessarily be satisfied through other means, if at all.

Unit-resolution is a special case of *resolution*. Suppose a formula $G$ contains the two clauses $[rM]$ and $[\overline{r}N]$. Resolution of these two clauses on the variable $r$ produces $[MN]$, which may be may be inserted into $G$ to produce $G'$. What makes this useful is that $G'$ is satisfiable if and only if $G$ is satisfiable. A proof of this fact follows.

Suppose $G$ has a satisfying truth assignment, $t$. If $r$ is $true$ under $t$, then the fact that $[\overline{r}N]$ is satisfied implies $t$ sets some literal from $N$ true. Thus $[MN]$ is satisfied. Similarly, if $r$ is $false$ then a literal from $M$ must be true, and again $[MN]$ is satisfied. In the other direction, because $G'$ subsumes $G$, a satisfying assignment for the former must satisfy the latter.

Once unit-subsumption and unit-resolution have been performed, if $F$ contains no clauses, it must be satisfiable. Therefore, the original CNF input to the algorithm is satisfiable. Otherwise steps P1 and P2 are repeated until there remain no unit clauses.

Step P3 purges every *pure literal* from the formula. A literal is pure in $F$ if it occurs in $F$ while its negation does not. It is safe to make these $true$ because if $F$ has a satisfying assignment, it has one which makes every pure literal $true$. Most SAT solvers do not actually implement this procedure, because it is prohibitively expensive to monitor how many instances of each literal are embedded in satisfied clauses, particularly if lazy BCP algorithms (as described in Chapter 3) are employed.

Once execution reaches step P4, no variable in $F$ is susceptible to removal through BCP or pure literal elimination. To continue the simplification process, some variable, $x$, is selected to be removed through other means. The method of selection, or *decision strategy*, plays an important role in determining the overall efficiency of the search. The strategy described in the listing above is only slightly different from the one in the original DP

listing in [12].

Now, $F$ is manipulated (without changing its satisfiability) to exclude the chosen variable, $x$. Using symbols as defined in the listing, $F$ is logically equivalent to $A \wedge B \wedge C$. Substituting for $A$ the logically equivalent expression $A' \vee x$, and for $B$ the logically equivalent expression $B' \vee \overline{x}$, yields $(A' \vee x) \wedge (B' \vee \overline{x}) \wedge C$. $F$ is unsatisfiable if and only if there is no satisfying truth assignment where $x$ is $true$, and no satisfying truth assignment where $x$ is $false$. So, $F$ is unsatisfiable if and only if $(B' \wedge C) \vee (A' \wedge C)$ is unsatisfiable. Thus $F$ is equivalent (with respect to satisfiability) to $(B' \vee A') \wedge C$, an expression in which $x$ no longer occurs. This expression is not necessarily a conjunction of clauses. In case it is not, a conversion is required before execution can return to step P0.

It is in this conversion that a weakness of the algorithm manifests itself. If $A'$ contains $n$ clauses, and $B'$ contains $m$, distribution of one over the other can produce as many as $mn$ clauses. There is potential for quadratic formula expansion each time a variable is removed in step P4.

Another way to see this is to understand that P4 amounts to reformulating $F$ as $D \wedge C$, in which $D$ is a conjunction of the clauses produced by resolving (on $x$) each clause in $A$ against every clause in $B$.

### 1.4.2 The Davis-Logemann-Loveland Algorithm

The Davis-Putnam algorithm has potential to produce exponential (in the number of variables) formula growth. In practice, although nothing approaching this worst case expansion is typically realized, the growth does pose a problem. For this reason, the DLL variation on the DP algorithm is preferred; it uses additional space only linear in the number of variables.

The input is a conjunction of clauses, $F$.

**L0.** If there exists no unit clause in $F$, go to step L2.

**L1.** Let $[x]$ be a unit clause in $F$. Remove from $F$ all clauses that contain the literal $x$. Remove the negation of $x$ from all clauses in which it appears. Go back to step L0.

**L2.** If $F$ contains an empty clause, return "unsatisfiable". Otherwise, while there is a pure literal in $F$, delete all clauses in which a pure literal is present. If $F$ is now empty, return "satisfiable".

**L3.** Select some variable $x$ that has a literal in $F$.

**L4.** Recurse on $F \wedge [x]$. If the call returns "satisfiable" then return "satisfiable".

**L5.** Recurse on $F \wedge [\overline{x}]$. If the call returns "satisfiable" then return "satisfiable", else return "unsatisfiable".

Steps L0 and L1 implement boolean constraint propagation. The BCP procedure as described in the previous section is only superficially different.

Step L2 determines first whether the formula contains an empty (unsatisfiable) clause. If it does not, then the formula may still be satisfiable, and pure literal elimination is performed. This may result in the deletion of all clauses from $F$, in which case the formula has been proven satisfiable. As discussed previously, a procedure to take advantage of pure literals may be (and in practice, usually is) omitted without affecting the correctness of the algorithm.

In step L3, a decision strategy is employed to choose $x$. As for the DP algorithm, the method of choosing variables for removal can have an important impact on overall efficiency. $F$ is satisfiable if and only if either it has a satisfying assignment under which $x$ is $true$, or it has one under which $x$ is $false$. Steps L4 and L5 test these two possibilities. $F \wedge [x]$ is $F$ altered so that BCP will force $x$ to be $true$. $F \wedge [\overline{x}]$ is $F$ altered so that BCP will set $x$ $false$. If $F$ is satisfiable under either restriction, it is satisfiable. If $F$ is not satisfiable under either restriction, it is unsatisfiable.

## 1.5 Resolution Refutations

A *general resolution refutation* (GRR) of CNF $F$ is a series $S$ of clauses $\{C_1, C_2, ..., C_n\}$ such that $C_n$ is empty, and every $C_k$ is either copied from $F$ or a product of the resolution of some pair $\{C_i, C_j\}$ for $i < j < k$. Clauses from $F$ may be copied into $S$ more than once. A *tree-like resolution refutation* (TRR) is a GRR, except each $C_k$ is restricted to be antecedent in at most one resolution step.

Every TRR is also a GRR. Furthermore, it has been proven there exist unsatisfiable formulas of size $n$ for which the shortest TRR is of size exponential in $n$, and yet the shortest GRR is of size polynomial in $n$. Therefore general resolution is strictly more powerful than tree-like resolution[5].

If DLL terminates and returns "satisfiable", it has discovered a satisfying truth assignment for $F$. It is easy to see that the assertions leading to an empty $F$ are available to

be gathered during execution, although the algorithm described above does not explicitly record them. If DLL terminates, returning "unsatisfiable", the particular pattern of execution that produced this result can be efficiently translated into a TRR of $F$. It is also true that any TRR for a CNF $F$ can be efficiently translated into a DLL refutation of $F$. In fact, translations in either direction are of linear complexity[15]. The refutation capacity of DLL is equivalent in both a theoretical and practical sense to that of tree-like resolution.

Although the solvers we study are derivative of DLL, they are at the same time fundamentally different. The essence of the difference is that every resolvent clause used in the refutation is also assimilated into the formula. With DLL, analogous information is present only implicitly, as the execution state of the program (e.g., which step is being executed by each recursive call). Because the solvers we study may reuse resolvents, the resolution proofs of unsatisfiability they produce need not be tree-like.

The DP algorithm does add resolvents explicitly. DP resolves clauses in steps P2 and P4: in P2, the unit resolutions of BCP; in P4, all possible resolutions on some variable $x$. Obviously missing is the flexibility of general resolution, and it has been proven that this constraint renders DP's proof efficiency stricly inferior to that of general resolution.

The solvers we study generate resolvents through a process that is not intrinsically incapable of any sequence of resolutions. Except for limits imposed by particular decision and restart strategies, the solvers have potential to produce any GRR[4]. One of our major concerns is taking advantage of this difference. Instead of attempting to backtrack exhaustively through the space of possibilities, we aim to accumulate clauses that will contribute to producing a refutation.

# Chapter 2

# Clause Learning

## 2.1 Overview

Let $F$ be a CNF over variables $v_1, ..., v_n$. $F$ may be understood in the obvious way as a Boolean function $f(v_1, ..., v_n)$. For our purposes, a clause may be added to $F$ as long as doing so does not change the corresponding function $f$. All clauses derivable from $F$ through a sequence of resolution steps meet this requirement. We are interested only in adding clauses that facilitate determination of satisfiability.

Various approaches to adding clauses have been suggested. Some are preprocessing methods: generally speaking, these work to augment the clause set through operations that are normally too costly to be beneficial if applied in the course of DLL. Search begins once preprocessing is complete. For example, *length-restricted resolution* resolves pairs of short clauses and accumulates (e.g.) unary, binary, and ternary resolvents. Empirical evidence indicates this is typically too expensive and not helpful enough to be worthwhile[26].

A more powerful preprocessing technique is *recursive learning* (RL). The concept relates closely to some ideas underlying Stålmarck's solver[32]. Short clauses from $F$ are considered; take $c$ to be such a clause. Any assignment that satisfies $F$ must make *true* some literal in $c$. For each literal $l$ of $c$, RL determines the conditions that must hold in case $l$ becomes *true*. All conditions found to hold for every $l$ individually must hold if $F$ is to be satisfied. Therefore $F$ may be further constrained to reflect the assertion of all such conditions. A natural suggestion is that RL could be used as a source of implications during search, but evidently this contributes too little to justify the overhead incurred[1].

In contrast to the foregoing, *conflict-driven clause learning* (CDCL) adds to the formula

during search, each time DLL empties a clause. CDCL represents a fundamental improvement over DLL, and all substantial recent advances in SAT solver technology are closely related to it. Relsat implemented a simple and effective version of the method[3]. A baroque and inefficient variant became the core of the GRASP solver[29]. Chaff coupled a technique used in GRASP with Relsat's simple overall approach to arrive at a scheme that has yet to be improved upon[30]. In this chapter, we describe the mechanics of CDCL. Then we explain why it is often better to learn a long clause when you could just as easily learn a short one instead.

## 2.2 DLL Revisited

Extend the term *unit clause* to encompass a disjunct in which one literal is free and all others are $false$. Boolean constraint propagation (BCP) sets the free literal of a unit clause $true$ until there exists no unit clause in $F$.

Henceforth, we assume pure literals are not eliminated.

By branching, DLL is able to try both assignments to each decision variable $x$. First, $x$ is set and the Boolean space under that restriction is explored. If and only if this search does not yield a satisfying assignment, $x$ is set the opposite way, and DLL operates on the resulting Boolean space. In case this second recursion fails also to produce a satisfying assignment, DLL returns that, if the formula is to be satisfied, an earlier restriction must be undone.

The *assignment stack* concept follows from the recursive character of the algorithm. When a variable takes on a truth value, a record of the event is pushed onto the assignment stack. When a call to DLL returns, the records it pushed are popped, and the corresponding variables become free.

The assignment stack can be viewed as a stack of *decision levels* (singular abbreviation, dl). Each dl is numbered according to its depth on the assignment stack: deepest is dl 0, second deepest is dl 1, and so forth. Decision level 0 is anomalous; it contains records of all assignments made prior to the first decision, i.e., products of BCP in the root DLL call. Every dl after 0 consists of (a) an assignment to exactly one decision variable, and (b) any assignments generated through BCP as a result of that decision.

CDCL appends clauses to the formula at the leaves of the DLL tree. We outline here an iterative form of DLL, extended to efficiently encompass CDCL and *nonchronological*

*backtracking* (NCB), both of which are covered in [3] and [29].

Input to the algorithm is a formula, $F$. The number of the current dl is stored in $y$.

**I0.** If $F$ contains an empty clause, return "unsatisfiable". Else, initialize $y$ to 0.

**I1.** Perform BCP. Push assignment records as part of dl $y$. If a clause is now all-$false$, go to step I4.

**I2.** If no variable is free, return "satisfiable".

**I3.** Increment $y$. Select some free literal and set it $true$. Push a record of this assignment to begin dl $y$. Go to step I1.

**I4.** If $y$ is 0, return "unsatisfiable".

**I5.** Derive from $F$ a set of implicates, $i$, with some element, $a$, an asserting clause (defined below). Revise $F$ to include $a$ and perhaps some other members of $i$.

**I6.** If $a$ has exactly one literal, let $y'$ be 0. Else, let $y'$ be the numeric label of the shallowest dl, deeper than dl $y$, at which some literal of $a$ became $false$. For every dl $> y'$, pop all records and unassign the variables they mention. Let $y$ equal $y'$. Go to I1.

In step I1, assignments are deduced through BCP and records are pushed onto the assignment stack. If there subsequently exists a clause consisting of $false$ literals only, a *conflict* condition has been reached. (A conflict condition will never hold prior to BCP.) In response to a conflict, execution continues at step I4.

On entry to I2, there is no conflict and BCP can deduce nothing more. If all variables are assigned, then $F$ has been satisfied, since all literals are either $true$ or $false$, and no clause contains $false$ literals only.

A decision is made in step I3. Some decision strategy is used to select a free literal. The literal is asserted, and a record of this event is pushed onto the assignment stack. This record is the first (i.e., deepest) element of a new dl. Execution returns to I1, where the dl will be completed through BCP.

The procedure for conflict handling begins in step I4. Let $c$ denote some particular clause containing $false$ literals only. There may be several such clauses. BCP deductions at decision level 0 are made in the absence of decision assignments. Thus, if $y$ is 0 then $c$ is falsified unconditionally and $F$ is unsatisfiable.

In practice, we do not actually compare $y$ against 0 each time an inconsistency arises. BCP at dl 0 occurs (a) before the first decision, and (b) each time a dl 0 unit clause is derived. It suffices to check for conflicts on these rare occasions and claim unsatisfiability if a conflict is found. This is more elegant, since these are dealt with as special cases anyway.

Step I5 is a main concern of this chapter. Implicates are generated by resolving an all-$false$ clause against clauses that became unit and played causative role in BCP assignments. Prior to the backtracking of step I6, all literals in these implicates are $false$. One or more implicates are added to the formula.

We require that some added clause contains exactly one literal whose variable became assigned at decision level $y$. The term *asserting* is used to refer to such clauses. Asserting clauses permit a simple coupling of CDCL and NCB.

Nonchronological backtracking takes place in step I6. First, the algorithm determines dl $y'$, the deepest dl at which $a$ is unit. If $a$ is independently unit, $y'$ is 0. Otherwise, the literals of $a$ are considered. Each was made $false$ at some decision level, and $y'$ is the largest numeric label $< y$ of a dl involved in this way.

Variables that became assigned within decision levels shallower than dl $y'$ are unbound, and dl $y'$ is exposed. Let $l$ denote the single literal from $a$ that was made $false$ in dl $y$. Since $l$ is now free, $a$ is unit. Execution returns to I1, where BCP sets $l$ $true$ and dl $y'$ is extended.

Typically, an efficient implementation diverges from this simplified form of the algorithm. Instead of requiring BCP to find and process $a$, $l$ is directly asserted. One benefit of this arrangement is that BCP always begins with a seed assignment. The initial population of dl 0 derives from singleton clauses of $F$. All BCP within any other dl is traceable back to a decision assignment. And BCP after backtracking must directly or indirectly stem from $l$. Propagation begins with clauses containing the negation of the seed.

The NCB procedure revokes decision levels between dl $y$ and dl $y'$ that did not contribute to the conflict. If the iterative DLL algorithm as a whole is suitably modified, it suffices to remove the shallowest dl involved in falsifying $a$. For example, see [29].

NCB as described in I6 has a heuristic aspect, since it may revoke in the absence of necessity. An interesting open question: is it beneficial to also remove noncontributing levels between dl $y'$ and dl 0?

NCB will not induce nontermination. The following constitutes a proof of that fact. If $F$ has $n$ variables, the assignment stack will never be more than $n + 1$ decision levels deep.

A *dl population vector* (DPV) is a sequence $\{c_0, c_1, c_2, ..., c_n\}$. Element $c_j$ is the number of assignment records in (i.e., the size of) dl $j$. If dl $j$ is not present, $c_j$ is 0.

For $F$, a DPV, $P$, is $>$ some other DPV, $Q$, if and only if there is an $i$ between 0 and $n$ (inclusive) such that $P_i > Q_i$, and $P_j = Q_j$ for all $0 \leq j < i$. That is, if and only if there is an $i$ such that, prior to element $i$, $P$ matches $Q$, and the $i^{th}$ element of $P$ is greater than the $i^{th}$ element of $Q$. This relation is obviously transitive.

The initial DPV is $\{0, 0, ..., 0\}$. It is invariant that, if the DPV changes from $P$ to $Q$ then $Q > P$. That is, the DPV magnitude increases monotonically. First, elements of the DPV are increased, and not decreased, by new assignments. Second, when there is backtracking from dl $y$ to dl $y'$, elements $c_y$ through $c_{y'+1}$ (inclusive) are zeroed. But, elements prior to $c_{y'}$ in the sequence are left unchanged, and $c_{y'}$ is increased. Thus the DPV is greater than it was before.

Since a finite number of variables implies a finite number of distinct DPVs, the algorithm must terminate. The greatest DPV is $\{n, 0, ..., 0\}$. Once all variables are assigned at dl 0, either there is a conflict and the algorithm returns "unsatisfiable", or there is no conflict and the algorithm returns "satisfiable".

## 2.3 Derivation of Implicates

Effective modern solvers work to derive, through resolution, short clauses. If the formula is unsatisfiable this tends toward building a refutation. If the formula is satisfiable, these clauses prune away vast solutionless spaces.

A machine with the capacity for nondeterministic computation may wield the full power of general resolution. Such a device may explore in parallel all binary resolution sequences. If a polynomial length GRR exists, it is discovered in polynomial time. For the moment this computer is unavailable, so we must resort to deterministic means. That this implies strictly reduced potential efficiency is unproven, but plausible and widely suspected.

Required is a proof search strategy to supplant the brute force of nondeterminism. Almost every complete algorithm used for SAT applies resolution guided by some such strategy. DP does unit resolutions and all possible resolutions on each decision variable. DLL works to implicitly construct a tree-like resolution refutation, the structure of which is dictated in large part by the decision heuristic. Width-bounded search[5] collects clauses, performing binary resolutions that have resolvents (a) not yet collected, and (b) of length

up to some bound. Upon deriving an empty clause, the procedure halts. When nothing else is possible, the bound is increased.

Conflict-driven clause learning is a resolution strategy. At each leaf in the search tree, CDCL resolves clauses that were involved in BCP along the path to that leaf. These clauses were made unit or $false$ by subsets of the same set of assignments. Thus the tendency is to resolve sets of compositionally similar clauses (i.e., clauses that share literals). Empirically, it appears this is a relatively good heuristic for many interesting formulas, in the sense that it facilitates derivation of useful clauses.

We now describe our refined implementation of the so-called *first-UIP learning scheme*, a simple and efficient means of deriving conflict implicates[41]. The main benefit of this scheme is that an asserting clause is arrived at through relatively few resolutions. The following are taken as input:

- CNF formula $F$. ($F_i$ denotes the $i^{th}$ clause of $F$.)
- Assignment stack $S$. ($S_i$ denotes the $i^{th}$ element of $S$. Deepest is $S_0$; shallowest is $S_T$. Each element is a $true$ literal.)
- Flag set $U$. (Associated with each variable, $v$, is a flag, $U_v$. If $v$ is assigned at dl 0, $U_v$ is initially $true$. Otherwise, $U_v$ is initially $false$.)
- Antecedent function $C$. ($C(v)$ is the index of the clause that became unit and was used to assign variable $v$ during BCP. Undefined for decision variables.)
- Level function $L$. ($L(v)$ is the number of the decision level at which variable $v$ became assigned. Undefined for free variables.)
- Clause index $X$. ($F_X$ is the all-$false$ clause from step I1.)

The output is $a$, an asserting clause. It is straightforward to adapt this procedure to select and return clauses intermediate in the production of $a$. For clarity, we present only the basic algorithm here.

Several integers are employed in coordinating the process. $i$ is the index of the clause being absorbed. $y$ is the numeric label of the shallowest (i.e., conflict) dl, a constant after step R0. $m$ keeps count of the variables from dl $y$ that are flagged, but have yet to be either resolved on, or included as a literal in $a$. $p$ is a pointer into the assignment stack, used for backward traversal.

**R0.** Set $a = []$, $i = X$, $m = 0$, and $p = T$. Set $y = L(v)$, where $v$ is the variable of $S_T$.

**R1.** For each literal $l$ in $F_i$:

[**R1a.**] Let $v$ be the variable of $l$. If $U_v = true$, skip R1b.

[**R1b.**] Set $U_v = true$. If $L(v) < y$, put $l$ into $a$. Else, set $m = m + 1$.

**R2.** Let $v$ be the variable of $S_p$. If $U_v = true$, go to R4.

**R3.** Set $p = p - 1$. Go to R2.

**R4.** If $m = 1$, put $\overline{S_p}$ into $a$ and return the result. Else, set $p = p - 1$.

**R5.** Set $m = m - 1$. Set $i = C(v)$. Go to R1.

Before explaining the above machinery, we clarify essentially what it operates to achieve. First, we prove $F_X$ contains at least two literals made $false$ within the conflict dl, $y$. We refer to these as *pivot literals*. If $F_X$ contains none, then the conflict occurred prior to dl $y$; contradiction. If $F_X$ contains exactly one, then $F_X$ was unit prior to dl $y$, and at that time BCP would have satisfied it or falsified it; contradiction.

Let clause $w$ initially be a copy of $F_X$. The procedure resolves pivot literals out of $w$ until exactly one remains. An iteration begins with the selection of the literal from $w$ whose negation appears shallowest in $S$. Necessarily, this is a pivot literal; call it $r$. It cannot be that $\overline{r}$ was a decision, since $w$ contains $\geq 2$ literals made $false$ in dl $y$, and a decision assignment is always the single deepest component of its dl. Thus there exists a clause, $F_c$, that became unit at dl $y$, causing BCP to assert $\overline{r}$.

Because $w$ contains $r$, and $F_c$ contains $\overline{r}$, the two clauses are resolvable. Furthermore, the resolvent is all-$false$, and so cannot be tautologous, since $w$ is all-$false$ and $F_c$ is all-$false$ except for $\overline{r}$. Let $w$ be replaced by this resolvent. The number of pivot literals in $w$ may have increased, decreased, or stayed the same, but certainly all the pivot literals were falsified earlier (i.e., deeper) than $r$.

If $w$ contains only one pivot literal, the process is complete. Otherwise, another iteration is needed. Because dl $y$ is of finite length, and all assignments it records are causally linked to a single decision, termination is inevitable.

Record $S_T$ is the literal on which BCP failed. Immediately after $S_T$ was pushed onto the stack and set $true$, BCP inspected clauses containing $\overline{S_T}$ to identify those made unit

by the assignment. In the process, $F_X$ was revealed to be all-$false$, and BCP halted. The details of breadth-first BCP are left for a later chapter.

Step R0 is responsible for initialization. $a$ begins empty and $i$ begins as a reference to $F_X$. No variables at the conflict dl have been flagged yet, so $m$ is zero. $p$ points to $S_T$. The numeric label of the conflict dl is determined, then stored to $y$.

The first time it is executed, step R1 simply recasts $F_X$. However, every subsequent execution of R1 corresponds to a resolution. Each literal $l$ from $F_i$ is considered. If $l$'s variable has been flagged, $l$ is ignored. Allowing R1a to bypass R1b serves to (a) ensure $m$ is incremented just once for each pivot, (b) prevent duplication of literals within $a$, and (c) filter out literals made $false$ in dl 0.

Otherwise, $l$'s variable is not yet flagged, and R1b is executed. Immediately, the flag is inverted. $l$ is a non-pivot literal if and only if it was made $false$ prior to dl $y$. If $l$ is a non-pivot literal, it will certainly be present in the output clause. So, it is put directly into $a$. Variables of non-pivot literals are flagged to ensure the procedure will never put one such literal into $a$ twice.

On the other hand, if $l$ is a pivot literal, it may later be resolved out. Therefore, step R1b does not insert pivots into $a$. Rather, $m$ is incremented to reflect the fact that the implicit intermediate clause now includes $l$. Pivot variables are flagged to distinguish in $S$ the negations of pivot literals, and to avoid overcounting.

Together, steps R2 and R3 implement a descent through dl $y$. Until $S_p$'s variable is found to be flagged, $p$ is decremented. When the branch to R4 is taken, $S_p$ is the shallowest negation of an unprocessed pivot literal. Since literals are arranged on the stack in the order they were asserted, any assignments causally involved in the assertion of $S_p$ are necessarily deeper on the stack than $S_p$ itself. Note that, the first time R2 is executed, $S_p$'s variable is found flagged, since $p = T$ and $\overline{S_T}$ is in $F_X$.

Step R4 checks $m$, the number of pivot literals in the implicit intermediate clause, including $\overline{S_p}$. If $m = 1$, the intermediate is now an asserting clause, so the single remaining pivot literal is inserted into $a$ and the procedure returns. (Note that if $v$ is a decision variable then $m = 1$.) Otherwise, $p$ is pointed deeper, past the current $S_p$, in preparation for the next execution of R2.

Step R5 decrements $m$ in anticipation of the pivot being resolved out. $i$ points to the antecedent clause of the pivot's negation (the clause that became unit, causing BCP to set the pivot literal $false$), and execution reenters R1.

Of course, it is inefficient to initialize $U$ before every call to the derivation procedure. In a good implementation, the set is persistent, maintained seperately. Once the CNF is available to the solver, all flags in $U$ are set $false$. Whenever some variable, $v$, becomes assigned at dl 0, $U_v$ is set permanently $true$. This allows unconditionally $false$ literals to be excluded from every new implicate, without requiring additional comparisons or complexity in the derivation code. (The implicate derivation routine is the second most expensive part of our solver, accounting for between 3 and 9 percent of the total runtime.) Flags set $true$ in step R1b are set $false$ before the derivation procedure returns. Pivot variables are unflagged as soon as they have been resolved out, i.e., after R1 completes. All the other temporary flags are cleared during R4, once $\overline{S_p}$ has been put into $a$. At that point, it suffices to clear flags for only those variables having a literal in $a$.

## 2.4 Participation Traces

CDCL solvers use BCP for two purposes simultaneously. First, BCP is used in the construction of exploratory truth assignments. This improves search efficiency by removing from consideration many non-satisfying points in the Boolean space.

Second, BCP is used to discover clusters of resolvable clauses. A heuristic benefit of this approach is that the clusters discovered tend to be populated by clauses that share literals. The clusters are made up of clauses involved in BCP along the path to a particular conflict. Loosely speaking, the more literals two clauses share, the more likely it is that they are both made unit or $false$ by a given truth assignment.

A CDCL participation trace, $P$, consists of a sequence of $n \geq 2$ clauses, $\{P_1, P_2, ..., P_n\}$. $P$ is a list of the clauses resolved together by CDCL to generate a particular asserting clause. Implicit in $P$ is a sequence of $n$ intermediate clauses, $\{m_1, m_2, ..., m_n\}$, where $m_1 = P_1$, and for all $i \in [2, n]$, clause $m_i$ is the resolvent of $P_i$ and $m_{i-1}$. The correspondence is, $P_1 = F_X$ and $m_n$ is the asserting clause.

*Trace redundancy* (TR) is a measure of literal duplication among clauses in a CDCL participation trace. The TR of trace $P$ is $d$ if and only if, on average, each literal occurring in $P$ does so as a member of $d$ distinct clauses. For a variety of non-random benchmarks, Table 2.1 reports the average TR over all derivations performed by our solver. Evidently, duplicates are abundant.

These non-random formulas encode modular structures, e.g., circuits. It could be argued

| **instance** | 7pipe | 9vliw_bp_mc | hanoi5 | 2bitadd_10 | longmult15 | c3540 |
|---|---|---|---|---|---|---|
| **average TR** | 2.31 | 2.50 | 3.10 | 3.75 | 3.29 | 2.56 |
| **average TL** | 29 | 21 | 15 | 14 | 92 | 45 |

Table 2.1: Trace redundancy (TR) and trace length (TL) for non-random instances

| **instance** | uuf100-01 | uuf250-01 | uuf250-02 | RTI_k3_n100_m429_0 |
|---|---|---|---|---|
| **average TR** | 2.65 | 2.57 | 2.51 | 2.61 |

Table 2.2: Trace redundancy for random instances

that, in such formulas, when clauses are resolvable, often they express constraints that involve many of the same variables; so, extensive literal duplication is to be expected, regardless of the resolution strategy employed. To illustrate that the trend holds even if modular structure is absent, we report average TR for random benchmarks in Table 2.2.

Included below are four participation traces produced in the course of solving some well-known benchmark instances[37][35][34]. These are typical traces, representative of the derivations performed by our CDCL solver. Clauses are delimited with square brackets. Each clause is numbered, to capture sequence of use. Comma-separated integers within a clause denote literals in the obvious way. The asserting implicate is labelled "resolvent".

```
 1. [1453,1465,1473,1505,1514,1635,1648,1656,1688,1696,-1725,-1749]
 2. [1595,1635,-1656]
 3. [1627,1679,-1688]
 4. [1453,1465,1473,1505,1534,-1595,-1603]
 5. [1453,1465,1473,1505,1514,1534,1587,1635,-1664,1668,-1679]
 6. [1603,1635,-1664]
 7. [1664,1696,-1725]
 8. [1675,1687,1725,-1786,-1797]
 9. [1675,1696,1749,-1797]
10. [1675,-1696,-1797]

resolvent: [1453,1465,1473,1505,1514,1534,1587,1627,
            1635,1648,1668,1675,1687,-1786,-1797]

 1. [469,484,493,501,509]
 2. [422,-469,-523]
 3. [431,-484,-523]
```

```
 4. [439,-493,-523]
 5. [455,-509,-523]
 6. [-422,-447]
 7. [-431,-447]
 8. [-439,-447]
 9. [-447,-455]
10. [447,482,-508]
11. [-501,-508,-523]
12. [-482,-523]

resolvent: [-508,-523]

 1. [-151,-636,-637]
 2. [636,697]
 3. [637,697]
 4. [-697,-727,-745]
 5. [696,727]
 6. [150,-600,-696]
 7. [-150,-151,-181]
 8. [745,775]
 9. [151,-229,-600,-744]
10. [25,-179,-229,-283,-298,-330,-331,-725,-775,-829,-844,-861,-883]
11. [181,-229]
12. [25,-179,229,-259,-298,-330,-600,-725,-744,-805,-844,-883]
13. [179,-629,-725]
14. [629,-725]
15. [25,-259,-283,-298,-330,-331,-600,725,-744,-805,-829,-844,-861,-883]
16. [-25,231,-247,-259,-283,-298,-330,-331,-600,-744,-805,-829,-844,-861,-883]
17. [777,805]
18. [231,247]
19. [231,259]
20. [231,-777]

resolvent: [231,-283,-298,-330,-331,-600,-744,-829,-844,-861,-883]

 1. [255,256,257,258,259,260,261,282,5171]
 2. [-261,-5155]
 3. [-260,-5155]
```

```
4. [-259,-5155]
5. [-258,-5155]
6. [-257,-5155]
7. [-256,-5155]
8. [-255,-5155]

resolvent: [282,-5155,5171]
```

Our solver implements a BCP policy favoring binary and ternary antecedents. This policy has a substantial positive impact, facilitating, for example, resolution patterns as exhibited in the latter trace.

## 2.5 Intermediates

The first-UIP learning scheme resolves until the intermediate clause contains only one literal set $false$ at the conflict decision level, dl $y$. Refer to this literal as $x$. When a literal is resolved out, it is replaced by the literals that made it $false$. Clearly, then, in combination with zero or more assignments from before dl $y$, $x$ being $false$ directly or indirectly caused every pivot literal in $F_X$ to become $false$.

Either $\overline{x}$ was a decision, or $\overline{x}$ was asserted through BCP. Suppose literal $d$ was the decision at dl $y$, and $d \neq \overline{x}$. Then, $d$ was involved in the conflict only insofar as it caused $\overline{x}$ to be asserted. All implications of $d$ that played a role in the conflict were implications of $\overline{x}$. If the decision at dl $y$ had been $\overline{x}$, rather than $d$, an identical conflict would have occurred. So, $\overline{x}$ is called the first-UIP, or *first unique implication point*.

Conflict-directed resolutions are often still possible once an asserting clause has been achieved. Unless only literals of decision variables are present, BCP antecedents are available, and CDCL may resolve them in. For example, if the pivot literal from an asserting clause became $false$ through BCP, CDCL can resolve it out. Then, it is always possible to produce a new asserting clause from the resulting intermediate. Of course, CDCL can also resolve out any non-pivot literal that was assigned through BCP.

Consider the *all-UIP learning scheme*[41]. The asserting clause produced by this method involves no more than one variable from any dl. Although it usually contains literals from a larger number of decision levels, the all-UIP clause is typically much shorter, on average, than the clause produced by the first-UIP scheme (see Table 2.3).

We include only a brief description of the all-UIP derivation procedure. Essentially, the first-UIP is isolated for each participating dl in turn, from shallowest to deepest. (No BCP antecedent resolution on variable $v$ ever introduces a literal that became assigned shallower on the stack than $v$.) Our implementation is a straightforward extension of our code for the first-UIP learning scheme.

Let clause $w$ initially be a copy of $F_X$. The following is repeated until the procedure returns. Take $y$ to be the label of the shallowest dl in which two or more literals from $w$ were made $false$. If there exists no such $y$, return $w$. Otherwise, resolve out of $w$ the literal that was falsified shallowest in dl $y$.

Our solver using the first-UIP learning scheme performs far better on many benchmark instances than our solver using the all-UIP learning scheme (see Table 2.5). Similar results have been published elsewhere. For example, several learning procedures were integrated into Chaff and experimentally evaluated in [41]. The first-UIP scheme was shown to be superior to various other schemes (every one of which applies more resolutions, on average, to produce each asserting clause).

It has been suggested that differences in learning scheme performance stem from differences in average cost of implicate derivation[2]. This is certainly wrong. First, even all-UIP learning code typically accounts for only a small fraction of total solver runtime. Second, compared to our first-UIP solver, our all-UIP solver consistently requires many more decisions and conflicts to solve a given instance.

Furthermore, the disparity is not a product of the learning-based decision strategies used in Chaff and its successors. These heuristics select variables that have literals in recently learned clauses. Different learning schemes yield different learned clauses. So, does the first-UIP scheme outperform the all-UIP scheme because it leads to better decisions? Evidently not. If branching decisions are made according to an arbitrary static variable ordering, the same pattern of learning scheme superiority holds[41].

The all-UIP scheme derives clauses that are shorter on average, but the first-UIP scheme is more successful at deriving the very short clauses that allow instances to be solved. Our conjecture is that the first-UIP learning scheme works relatively well because it uses fewer resolutions to generate the clauses that are appended to the formula (see Table 2.4). Fewer resolutions per implicate implies greater proof-search flexibility. The larger the average number of resolutions each added clause represents, the larger the number of unavailable derivation intermediates. Such intermediates are potentially more useful than the clause

actually added to the formula. For example, the first-UIP implicate is an intermediate in the production of the all-UIP implicate.

The published experimental results, all of which we have independently reproduced, may be interpreted as showing a strong negative correlation between average resolution counts and solver performance. Broadly, holding the input formula constant, the more resolutions that go into producing each implicate, the worse the solver performs. This suggests that it might be worthwhile to pursue a reduction in the coupling between learning and nonchronological backtracking. Restricting CDCL to produce an asserting clause forces it to resolve until an asserting clause is produced. Perhaps the solver should also learn clauses that serve no backtracking function.

The asserting clauses generated by CDCL aid in the search for a satisfying assignment by increasing the deductive power of BCP, and by facilitating NCB. There is a separate, but closely related, need for some of these implicates to be useful in the derivation of short clauses. The latter make explicit simple patterns present implicitly in the formula. It is by working with the explicit representations of these patterns that modern solvers are able to determine the satisfiability of enormous, but structurally simple, formulas.

A CDCL derivation begins with a falsified clause, $F_X$. Resolutions are applied until an asserting clause, $c$, is produced. The first intermediate is $F_X$. The second intermediate is the resolvent of $F_X$ against a BCP antecedent. The third intermediate is the resolvent of the second intermediate against a BCP antecedent. And so on, with the final resolvent being the last intermediate, $c$. With the exception of $F_X$, no intermediate in a CDCL derivation is already present in the formula at the time the derivation is undertaken. So, we may append any intermediate, except the first, onto the formula, without introducing a duplicate clause. This is proven in Chapter 3.

An intermediate clause may be extracted between steps R1 and R2 in our implementation of the first-UIP learning scheme. Simply, append to $a$ the negations of the flagged literals in dl $y$ that have stack positions $\leq p$, and store the result.

The first-UIP learning scheme derives an asserting clause through the minimum sufficient number of antecedent resolutions. Although this minimum tends to be low on average, it is occasionally high. In these degenerate cases, intermediate clauses are often shorter than the final resolvent. Also, the set of variables represented in an early intermediate usually bears little resemblance to the set represented in the asserting clause. (When the all-UIP scheme is applied, it is common for the set of represented variables to change completely, or almost

| instance | first-UIP | all-UIP | instance | first-UIP | all-UIP |
|---|---|---|---|---|---|
| dlx2_cc[36] | 33 | 13 | logistics.c[22] | 6 | 5 |
| 3pipe[37] | 78 | 17 | 2bitadd_10[10] | 36 | 13 |
| 6pipe[37] | 495 | 28 | avg-check-5-35[34] | 61 | 16 |
| 7pipe[37] | 680 | 27 | 9vliw_bp_mc[37] | 187 | 19 |
| 7pipe_bug[37] | 236 | 26 | longmult15[6] | 104 | 11 |
| c3540[28] | 97 | 17 | barrel9[6] | 119 | 27 |
| c7552[28] | 58 | 50 | hanoi6[34] | 44 | 21 |

Table 2.3: Average literals per implicate added: first-UIP vs. all-UIP

| instance | first-UIP | all-UIP | instance | first-UIP | all-UIP |
|---|---|---|---|---|---|
| dlx2_cc[36] | 16 | 48 | logistics.c[22] | 5 | 13 |
| 3pipe[37] | 15 | 69 | 2bitadd_10[10] | 14 | 55 |
| 6pipe[37] | 23 | 163 | avg-check-5-35[34] | 15 | 48 |
| 7pipe[37] | 29 | 185 | 9vliw_bp_mc[37] | 21 | 102 |
| 7pipe_bug[37] | 21 | 163 | longmult15[6] | 92 | 675 |
| c3540[28] | 45 | 876 | barrel9[6] | 55 | 2507 |
| c7552[28] | 36 | 2525 | hanoi6[34] | 20 | 103 |

Table 2.4: Average resolutions per implicate added: first-UIP vs. all-UIP

| instance | first-UIP | all-UIP | instance | first-UIP | all-UIP |
|---|---|---|---|---|---|
| dlx2_cc[36] | 1 | 211 | logistics.c[22] | 1 | 1 |
| 3pipe[37] | 1 | >3600 | 2bitadd_10[10] | 341 | 945 |
| 6pipe[37] | 73 | >3600 | avg-check-5-35[34] | 244 | 2318 |
| 7pipe[37] | 282 | >3600 | 9vliw_bp_mc[37] | 53 | >3600 |
| 7pipe_bug[37] | 8 | 170 | longmult15[6] | 236 | 642 |
| c3540[28] | 28 | >3600 | barrel9[6] | 41 | >3600 |
| c7552[28] | 14 | >3600 | hanoi6[34] | 760 | >3600 |

Table 2.5: Solver runtime in seconds on a SunBlade1000 750MHz Ultra I

completely, over the course of a derivation.) It may be beneficial to append intermediates onto the formula, along with the asserting clause. This must be done sparingly, since BCP is more expensive on a formula that contains more clauses.

We have only preliminary experimental results in this area. For example, adding the second intermediate allows our solver to prove 3pipe unsatisfiable in less than two minutes, using a static decision strategy. If only the first-UIP asserting clause is added, more than 20 hours are required. However, it is not yet clear that such an approach is useful over a wide range of interesting formulas. There are many examples of instances for which storing the second intermediate appears to provide no benefit. Further research along these lines seems to be justified.

## 2.6 Omissions

Several important aspects of modern solvers have not been covered at length in this thesis. Although clause deletion and search restarts are used in our solver, our implementation is standard and unremarkable. We refer the reader to [30] for an overview of the methods we have adopted.

Memory constraints and BCP cost dictate that the formula cannot be allowed grow without bound. Learned clauses must be deleted as they become too numerous. Periodically, our solver deletes a random selection of learned clauses that exceed some threshold length. Extremely long clauses are deleted during backtracking, as in Chaff.

Our solver restarts periodically, every 16,000 conflicts. When the solver restarts, every variable that has become assigned as a direct or indirect result of a decision assignment is set $free$. Iterative DLL resumes in step I3 at decision level 0. Restarting acts to change the space in which the solver is searching for a satisfying assignment. It also tends to change the set of clauses that CDCL resolves.

# Chapter 3

# Boolean Constraint Propagation

## 3.1 Overview

As first emphasized in the Chaff literature, typically about 90 percent of the runtime of a clause-learning DLL solver is spent performing BCP[30]. There are two reasons for this. First, the BCP procedure is executed frequently, every time a variable is assigned a truth value. Second, BCP operates both broadly and nonsequentially over a data structure that is normally many times larger than the L2 cache of the host computer. This diffuse access pattern means data cache is frequently missed. When L1 and L2 are missed, a cache line is brought in from main memory (assuming no L3). Because of the gap between processor speed and main memory access speed, this fetch is a bottleneck, very expensive relative to other solver procedures, which saturate the CPU. The extraordinary cost of BCP stems from a memory latency problem.

Our focus is the *two watched literals* (TWL) BCP algorithm introduced in Chaff[30]. TWL is a simplification of the *head/tail lists* (HTL) algorithm[39] introduced in SATO[40]. The main benefit of these two-pointer schemes is that they load fewer cache lines than, e.g., the counter-based schemes in Relsat and GRASP. Although an elaborate HTL implementation can have superior expected-case abstract efficiency characteristics, HTL is reliably outperformed in practice by TWL.

We detail breadth-first BCP and prove a property used in Chapter 2. Then, we explain TWL and describe the various refinements that allow our solver to perform BCP faster than any other solver publically available.

## 3.2 Breadth-first BCP

To begin, we present the basic algorithm, abstracting away the details of the link between a literal and the clauses that contain its negation. For the moment, the reader may assume that each literal, $l$, is linked to every clause in which $\overline{l}$ occurs.

We assume a queue to store and dispense (literal, antecedent) pairs in FIFO order. The procedure requires access to the formula, $F$; the assignment stack, $S$; and the structure used to store variable information, $V$.

The input is a seed literal, $d$; an antecedent reference, $a$; and the numeric label of a decision level, $y$. If $d$ was a decision, $a$ is null and $y$ is the index of the dl that $d$ has been selected to originate. Otherwise, the solver has just backtracked, $a$ is the index of the newly derived clause, and $y$ is the deepest dl at which $a$ is unit.

Two pieces of memory are used within the procedure for coordination. The first is a literal, $l$, and the second is an antecedent clause reference, $c$.

**B0.** Set $l = d$, and $c = a$. The queue is initially empty.

**B1.** Write to $V$ that $l$ is $true$ at dl $y$ because of $c$. Push $l$ onto $S$.

**B2.** For each clause, $x$, in $F$, to which $l$ has a link:

[**B2a.**] If $x$ contains a $true$ literal or $\geq 2$ free literals, skip B2b.

[**B2b.**] If $x$ contains a free literal, $t$, enqueue $(t,x)$. Else, return "conflict at $x$".

**B3.** If the queue is empty, return "no conflict".

**B4.** Dequeue one pair, assign it to $(l,c)$.

**B5.** If $l$ is $true$, go to B3. Else, go to B1.

The seed literal is free when it is passed into the BCP procedure. It becomes assigned in step B1, and then is pushed onto the assignment stack. If the seed was a decision, this begins dl $y$. Else, this begins an extension of dl $y$ made possible by the new implicate, $a$.

In step B2, the procedure visits all clauses to which $l$ is linked. Every one of these clauses contains $\overline{l}$. It is absolutely essential that $l$ have a link to every clause that could become unit or $false$ as an immediate result of $l$ becoming $true$. Two-pointer algorithms, like HTL and TWL, simply minimize the number of links. They permit $l$ to not be linked to some clauses, even though the clauses contain $\overline{l}$.

Step B2a identifies clauses that are inconsequential, neither unit nor $false$. In step B2b, $x$ is not satisfied and it contains fewer than two free literals. If $x$ has one free literal, $t$, then $t$ must be $true$ if $F$ is to be satisfied. So, $t$ is scheduled to become assigned, with $x$ as its antecedent clause.

If $x$ is all-$false$, step B2b halts the process. In general, the following must occur for a conflict to arise. Some suitable literal, $z$, whose negation is an element of $x$, is enqueued prior to $l$. Before, or when, $z$ is dequeued and asserted, $l$ is enqueued. Subsequently, as a result of $z$ being dequeued and asserted, $x$ becomes unit, and so $\overline{l}$ is enqueued. Finally, when $l$ is dequeued and asserted, $x$ is falsified.

If no assignments are pending in step B3, then BCP halts, without having produced a conflict. Otherwise, the oldest pair is dequeued, overwriting $l$ and $c$. If the implied literal, $l$, is already $true$, then the implication is redundant, so the pair is ignored. Else, $l$ is asserted and the consequences are determined.

Consider the following change to the above algorithm. As soon as $t$ is known to be the only free literal in clause $x$, $t$ is made $true$ and is pushed onto the assignment stack. That is, instead of enqueuing $(t,x)$, $t$ is asserted and pushed, as in step B1. This allows BCP to continue after falsifying a clause, but conflict is detected eventually whenever it occurs. A separate queue is not necessary. It suffices to read literals off $S$, stopping once all those shallower than $d$ have been processed. No literal enters the stack twice during one execution of the procedure. The modified algorithm is simpler and faster.

Furthermore, it forces the same set of literals as breadth-first BCP. The order in which assertions are made is immaterial to BCP correctness and power. Depth-first, breadth-first, arbitrary: sequence has no bearing on the set of literals deduced.

But, solver performance is slightly impaired on some CNFs, relative to when breadth-first BCP is applied. Apparently, the problem is that the simplified algorithm asserts earlier than the breadth-first algorithm. This seems to result in selection of longer antecedents, conflicts that involve more variables, and so on.

## 3.3 Intermediates

With the obvious exception of the initially falsified clause, $x$, no intermediate in a CDCL derivation is subsumed by a clause that is present in the formula when the derivation begins. A proof follows.

Every intermediate, $i$, except $x$, is the resolvent in a CDCL resolution. Every such resolution involves some other all-$false$ intermediate, $w$. (It may be that $w$ is $x$.) CDCL resolves on $z$, the shallowest literal in $w$, using $a$, the antecedent clause for $\overline{z}$. The resolution of $w$ and $a$ produces $i$.

Without loss of generality, take $w$ to be $[zJ]$. Set $J$ consists of one or more literals, each of which was made $false$ earlier than $z$. Similarly, take $a$ to be $[\overline{z}K]$. Set $K$ consists of literals that became $false$ before $\overline{z}$ became $true$.

Suppose some clause, $u \in F$, subsumes $i$. When $z$ was set $false$, step B1 was executed with $l = \overline{z}$. Immediately before $\overline{z}$ became $true$, both $J$ and $K$ must have been all-$false$. Thus, $u \subseteq i = [JK]$ must have been falsified by some earlier assertion. However, BCP always halts during step B2 after it makes an assertion in step B1 that falsifies a clause. Contradiction.

## 3.4 The Two Watched Literals Algorithm

We assume that a clause is realized as an array of literal instances. Neither unit clauses nor empty clauses are represented. The first and last literals in every clause's array are sentinels that evaluate $true$ or free.

A literal instance has three fields: the sign flag, the watched flag, and the variable index. The sign flag is one bit, indicating whether or not the variable is negated. The watched flag is one bit, indicating whether or not there exists a watch structure that points to the instance. The variable index uniquely identifies the variable of the literal. No clause contains two non-sentinel literal instances that have the same variable index.

A watch structure has two fields: the direction flag, and the literal instance pointer. The direction flag is one bit, indicating either that the watch has been moving toward the first literal in the clause, or that it has been moving toward the last literal in the clause. The instance pointer is the memory address of a particular literal instance within a clause. Each variable is associated with two lists of watch structures. List $W(l)$ contains all watch structures that point to instances of literal $l$.

For every variable, $v$, both $W(v)$ and $W(\overline{v})$ begin empty. For every literal instance, the watched flag begins $false$. Then, two watch structures are added per clause, and all pointed-to literal instances have their watched flags inverted. In detail: given a clause of length $n$, $[s_a l_1 ... l_n s_b]$, where $s_a$ and $s_b$ are sentinels; any two literal instances, $l_i$ and $l_j$, are

selected, such that $i, j \in [1, n]$ and $i \neq j$. A watch structure, with an arbitrary direction flag and a pointer to $l_i$, is appended to $W(l_i)$. Similarly for $l_j$. Finally, instances $l_i$ and $l_j$ have their watched flags set $true$.

To integrate TWL data structures into the breadth-first BCP algorithm of section 3.2, replace step B2 with step W0, below. In the following, watch structures are represented by (direction flag, pointer) pairs. Literal instances are represented by (sign flag, watched flag, variable index) triples. A (sign flag, variable index) pair is translated into a literal in the obvious way, taking, say, a sign flag of $true$ to imply negation.

**W0.** For each watch structure, $w = (r, p) \in W(\overline{l})$:

[**W0a.**] Let $q = p$, $e = 0$, and $o = null$. $x$ is not initialized.

[**W0b.**] If $r = 0$ then $p = p - 1$, else $p = p + 1$. Let $(s, t, v)$ be the instance at $p$.

[**W0c.**] Let literal $u$ be $(s, v)$. If $u$ is $false$ then go to W0b.

[**W0d.**] If $u$ is not a sentinel, go to W0i.

[**W0e.**] If $r = 0$, then let $x = p$.

[**W0f.**] If $e = 0$, then let $e = 1$, $p = q$, $r = 1 - r$; then go to W0b.

[**W0g.**] If $o = null$, then return "conflict at clause $x$".

[**W0h.**] If $o$ is free, then enqueue ($o$,$x$). Skip W0i, W0j, and W0k.

[**W0i.**] If $t$ is $true$, then let $o = u$ and go to W0b.

[**W0j.**] Remove $w$ from $W(\overline{l})$. Set the watched flag $false$ at $q$.

[**W0k.**] Append ($r$,$p$) to $W(u)$. Set the watched flag $true$ at $p$.

The procedure works with each element of $W(\overline{l})$ in turn. Each of these watch structures is a BCP link from $l$ to a clause in $F$. If some clause, $c$, is susceptible to immediately become unit or $false$ when $l$ is asserted, then there is a watch in $W(\overline{l})$ that points into $c$. TWL succeeds by maintaining the condition that no $false$ literal is watched in any clause that contains an unwatched literal that is $true$ or free. When $l$ is $true$, instances of $\overline{l}$ cannot be watched, except in clauses where all unwatched literals are $false$.

The procedure visits every clause that contains a watched instance of $\overline{l}$. Each of these clauses is visited with the intent to find an unwatched literal, $x$, that is $true$ or free. If

such an $x$ is found, a new watch structure is added to $W(x)$, and the current watch on $\overline{l}$ is removed from $W(\overline{l})$. So, there remain two watch structures per clause.

If no such $x$ is found, every unwatched literal in the clause must already be $false$. In this case, the watch on $\overline{l}$ persists. The clause is satisfied, unit, or $false$, depending on the status of the other watched literal, $k$. If $k$ is $false$, the clause is $false$. If $k$ is $true$, the clause is satisfied. If $k$ is free, the clause became unit as $l$ became $true$.

Within the breadth-first BCP framework, this ensures that all unit clauses are detected as they arise. If a clause becomes unit, there is an assignment responsible for the transition. Immediately prior to the transition, the clause has exactly two free literals, both of which must be watched. After one of the two becomes $false$ in step B1, step W0 (standing in for step B2) must fail to find an unwatched non-$false$ literal in the clause. Therefore, detection. A similar argument may be used for clauses that become falsified.

TWL adjusts pointers only during BCP. Watch structures need not be manipulated during backtracking. In contrast, the head/tail lists (HTL) algorithm often needs to adjust pointers during backtracking. A good HTL implementation incurs only amortized O(1) backtracking cost, since pointer movement during an ascent through the search tree is simply a reversal of the pointer movement performed during descent. However, the cost of visiting clauses is enormous in practice, since L2 cache is frequently missed. Thus, while backtracking inflicts O(1) overhead for both HTL and TWL, the constant for HTL is significantly larger.

Backtracking frees assigned variables. It never violates the TWL requirement, i.e., that no $false$ literal be watched in any clause that contains an unwatched non-$false$ literal. Suppose that, before backtracking, a $false$ literal, $x$, is watched in clause $c$. Then all unwatched literals in $c$ are $false$, and none became $false$ shallower on the assignment stack than $x$. This holds because it must be that all unwatched literals in $c$ were $false$ when $x$ became $false$. Otherwise, the watch on $x$ would have been replaced by a watch on a non-$false$ literal in $c$. During backtracking, variables are unassigned in the same order their literals are popped off the assignment stack. Thus, for any particular clause, backtracking frees the watched $false$ literals before it frees the unwatched $false$ literals.

In step W0a, memory is initialized. First, $p$ is copied to $q$, so that both $p$ and $q$ point to the same watched instance of $\overline{l}$. Since $p$ is altered in step W0b, $q$ is kept to support efficient inversion of the watched flag on $\overline{l}$, which occurs when some other literal becomes watched. Also, $q$ is used as a starting point for movement in the opposite direction when search for a

non-$false$ literal runs off one end of the clause (i.e., encounters a sentinel).

Second, $e$ is zeroed. $e$ records the number of sentinel encounters. Once the search has encountered both sentinels, the entire clause has been explored, and every one of its unwatched literals is known to be $false$. On the first sentinel hit, $e$ is incremented from 0 to 1. Then, if a sentinel is hit while $e$ is 1, the clause is all-$false$, except for, perhaps, the other watched literal. We use the term *active* to describe such a clause.

Third, literal instance $o$ is initialized to null. $o$ will store the other watched literal so that it may be inspected efficiently if the clause is found to be active. The TWL algorithm must fully explore a clause before declaring it active. Therefore, if the clause is declared active, the other watched literal, $k$, was encountered during the search. If $k$ is $false$, it is not copied, and $o$ remains null. Otherwise, $o = k$. Using $o$, the procedure can quickly determine whether an active clause is falsified, satisfied, or unit.

Finally, pointer $x$ is declared, but not initialized. $x$ is used to record the first memory address of the clause. This information facilitates access to the clause's literals during conflict-driven clause learning.

Step W0b alters the content of $p$, depending on $w$'s direction flag, $r$. If $r = 0$, $p$ is made to point at the instance preceding the instance at which it currently points. If $r = 1$, $p$ is moved to the next instance in the opposite direction. The sign flag, watched flag, and variable index of the newly pointed-to instance are cached in $s$, $t$, and $v$, respectively.

In W0c, literal $u$ is derived from sign $s$ and variable index $v$. If $u$ is $false$, $p$ is neither pointing at a sentinel, nor pointing at a literal that can become watched; execution returns to W0b. This tight loop searches the clause for a non-$false$ literal. Note that a good implementation does not actually use a comparison in step W0b.

Sentinel literals are perpetually non-$false$. Furthermore, they are easily distinguishable from non-sentinel literals. Suppose some variable, $h$, does not participate in the CNF. If variable $h$ is kept either $true$ or free, literal $h$ may be used as a sentinel. Step W0d checks whether $u$ is a sentinel. If it is, execution continues into step W0e. Otherwise, it may be possible to watch $u$ instead of $\overline{l}$; execution jumps to W0i.

In step W0e, search has reached a clause boundary sentinel. If $r = 0$, it is the low memory boundary that has been hit, so $p$ is stored to $x$. The procedure uses $x$ only after determining the clause is active. To determine the clause is active, TWL must encounter both boundaries. Therefore, $x$ is always initialized before use.

If $e \neq 0$ in step W0f, the search has reached both boundaries. Therefore, all unwatched

literals in the clause are $false$. Execution falls through into step W0g. Otherwise, only one boundary has been reached. Execution reenters step W0b; search resumes at $\overline{l}$, but in the opposite direction.

If $o$ is null in step W0g, both watched literals are $false$. In step W0i, $o$ becomes a copy of the literal instance at $p$, if the watched flag at $p$ is $true$. But, step W0i is never executed for a $false$ literal. The loop over W0b and W0c breaks only when $p$ points to a literal that is either $true$ or free.

(Suppose clause $c$ has two $false$ watched literals, $l_1$ and $l_2$. Further, suppose $l_1$ was set before $l_2$. It must be that falsifying $l_2$ falsified $c$: if $l_1$ is both $false$ and watched, then $c$ is active and $l_2$ must already be queued for assertion. Thus, a clause may be declared $false$ immediately if the W0b/W0c loop encounters a $false$ watched literal.)

Otherwise, the clause is active and the second watched literal is non-$false$. If $o$ is $true$, the clause is satisfied. Else, $o$ is queued to be set $true$ with antecedent $x$. In both cases, control returns to the outermost loop.

If $u$ is neither $false$ nor a sentinel, step W0i is executed. $p$ points to an instance of literal $u$ that has watched flag $t$. If $t$ is $true$, the literal is already being watched, so $u$ is copied and the search loop is reentered. Otherwise, the literal at $p$ is available to be watched in place of $\overline{l}$. Steps W0j and W0k implement the transition from watching $\overline{l}$ to watching $u$. The watched flag on $\overline{l}$ is set $false$. The watched flag on $u$ is set $true$. The current $\overline{l}$ watch structure is deleted and a new watch structure is added to $W(u)$.

The new watch structure takes on the current direction flag, $r$. This guides future search (when $u$ is set $false$) away from the segment of the clause that was just explored. If a clause is visited several times between backtracks, this tends to slightly reduce the average time spent searching for a literal to watch.

## 3.5 Refinements

A typical CPU operates only on data in its registers. If a machine word is to be, e.g., compared, it must first be copied into a register on the chip. Suppose a program operates on the word at main memory address $x$. A load instruction is used to copy the appropriate word into a register. If a duplicate of the word at address $x$ is already present in the L1 cache, the load finishes in one or two CPU cycles. If not, the program has suffered an L1 miss. In response, the L2 cache is probed. If the word at $x$ is cached in L2, the load completes in

roughly five to ten cycles. Otherwise, an L2 cache miss occurs, and the word is copied in from main memory at a cost of 50 to 250 cycles[21]. (This is a simplified description of a very complex system. We are primarily concerned with broad trends in cost.)

Memory is copied through the cache hierarchy in blocks, called *cache lines*. A cache line is a series of consecutively addressed bytes. In most modern computer architectures, cache lines are 32 bytes long (although the L1 line length may differ from the L2 line length[21]). For lines of length $L$, two addresses, $a_1$ and $a_2$, are on the same cache line if and only if $\lfloor a_1/L \rfloor = \lfloor a_2/L \rfloor$. When a byte is pulled into cache, every other byte on the same line is also pulled in. If L1 is missed but L2 is hit, a cache line is copied from L2 into L1. If L2 is missed, a cache line is copied from main memory into both L1 and L2.

It is not uncommon for industrial formulas to contain hundreds of thousands of clauses, and millions of literal instances. And, clause learning leads to rapid formula growth. Every leaf in the DLL search tree induces an implicate. Frequently, these implicates are hundreds of literals long. Formulas expand to be many tens or hundreds of megabytes in size. In contrast, a Pentium 4 chip has 8 kilobytes of L1 for data, and 512 kilobytes of L2[21].

The disparity between cache size and formula size means that only a tiny fraction of the formula is ever cached. Furthermore, assignments tend to impact many clauses. Suppose a formula contains $i$ literal instances, over $v$ variables. Since no clause contains two identical literals, setting a literal $false$ shortens $i/2v$ clauses, on average. Even for counter-based schemes, where shortening a clause usually only involves decrementing a counter, clause updates frequently miss cache. There are so many clauses that just a 4-byte record for each amounts to a structure that does not fit in the cache. And, clause traversals import numerous lines, polluting the cache, displacing other data. Recent additions are hit repeatedly during sequential clause traversals, but otherwise, lines tend to be displaced from cache before being reused. This is true of all known BCP schemes.

Two-pointer BCP algorithms are more efficient than counter-based BCP algorithms, since minimizing the number of links from literals to clauses reduces the number of clause visits during BCP. Both HTL and TWL maintain just two links per clause. A clause is visited during BCP if and only if one of its watched literals is made $false$. If literal $l$ is not watched in clause $c$, clause $c$ is not visited when $l$ becomes $false$.

Regardless of memory latency, TWL is cheaper than counter-based BCP algorithms. Fewer links implies a reduced average amount of work per assignment, and no computation is necessary during backtracking. Counter-based BCP is simple, but it imposes a much

heavier CPU load than TWL.

However, the primary strength of TWL lies in its memory access pattern. It is expensive to visit clauses, because doing so frequently results in an L2 cache miss. Having two links per clause reduces the average number of visits triggered by each assignment, and no clause is visited during backtracking. On the other hand, it is relatively inexpensive to traverse a clause upon visiting it, because contiguous reads put cache lines to good use. Most clause traversals complete within a single cache line, and therefore hit L1 every time a literal is loaded. In an average clause traversal, step W0b is executed less than three times.

### 3.5.1 Binary Clause BCP

Here, we apply the term *binary clause* to refer to clauses with exactly two literals, $false$ literals included. Constraint propagation through binary clauses is particularly simple. If the formula contains $[l_0 l_1]$, then for $x \in \{0, 1\}$, if $l_x$ is $false$, $l_{1-x}$ must be $true$. It is inefficient to use a fully general BCP mechanism to capture this. In binary clauses, TWL is at its most degenerate, since every literal must be watched. Boundary sentinels double the memory footprint of each clause, and the entire footprint is traversed on every visit. Attempts to find another literal to watch are wasted effort. The inefficiency is significant because binary clauses are abundant in the formulas we are concerned with solving. In standard bounded model checking benchmarks[38] more than 70 percent of the clauses are binary. More than 90 percent are binary in standard microprocessor verification benchmarks[37].

A better method relies on a specialized binary clause representation. Each variable, $v$, is associated with two lists of literals, $B(v)$ and $B(\overline{v})$. Each clause, $[l_0 l_1]$, is represented by a pair of list entries: an instance of $l_1$ in $B(l_0)$, and an instance of $l_0$ in $B(l_1)$. If a literal, $l$, becomes $true$, all literals in $B(\overline{l})$ are implied.

Binary clause BCP for $l$ occurs in a single pass through $B(\overline{l})$. For each literal $x \in B(\overline{l})$: If $x$ is $true$, then $x$ is ignored. If $x$ is $false$, there is a conflict, and falsified clause $[\overline{l}x]$ is returned explicitly. Otherwise, $x$ is free, so $x$ is queued for assertion. In the queue, $x$ is paired with $\overline{l}$, rather than a pointer to an antecedent clause. Antecedent records are flagged to indicate their correct interpretation, as a literal or as a pointer. The CDCL procedure is readily adapted to distinguish and handle both cases. In particular, note that the literal $\overline{l}$ is sufficient antecedent information to support a CDCL resolution on $x$.

Together, TWL and the above procedure implement step B2 of the breadth-first BCP algorithm. Once $l$ is assigned in step B1, binary BCP is performed via $B(\overline{l})$. Then, TWL

performs non-binary BCP via $W(\bar{l})$. Sequencing BCP this way favors binary antecedents, and so promotes binary clause resolutions in CDCL derivations.

The described method is highly efficient. First, if $B(\bar{l})$ is implemented as a contiguous array, the memory access pattern of a natural traversal is ideal. In contrast, suppose each binary clause is represented as a pair of adjacent literal instances. Each clause is twice the size, so half as many fit on a cache line. TWL boundary sentinels aggravate this. Also, clauses that contain $\bar{l}$ are not necessarily grouped together in memory. Worst case, it could be that no two are on the same cache line together.

Second, the traversal procedure can be very computationally inexpensive. Usually, when BCP visits a binary clause, the implied literal is already $true$. The procedure first checks whether the current element of $B(\bar{l})$ is $true$. If it is, a clause has been dispensed with in a single comparison.

Also, bounds checking overhead is easily cut from the loop that iterates through $B(\bar{l})$. If a $false$ sentinel literal is appended to each list, the bounds test may be embedded in the conflict code. Whenever a $false$ literal is encountered in $B(\bar{l})$, a test is done to determine whether or not the literal is a sentinel. Because real conflict is relatively rare, and the procedure is so simple, the savings are significant.

(Tangentially, we note here a useful property of TWL. Suppose a clause consists of $n$ literals. Further, suppose two of them are free, while the other $n-2$ are $false$. We term such clauses *conditionally binary*. When BCP finishes without conflict, both free literals must be watched in every clause that is conditionally binary. Therefore, if a literal, $l$, is free in a conditionally binary clause, $c$, then $W(l)$ contains a watch structure that points into $c$. To determine how many conditionally binary clauses $l$ participates in, it suffices to count how many clauses referenced in $W(l)$ are conditionally binary.)

### 3.5.2 Ternary Clause BCP

In most of the formulas we are concerned with, ternary clauses are much less common than binary clauses. There are some exceptions, e.g., miters for circuit equivalence checking[33]. Ternary clauses appear frequently in formulas derived via basic translation from logic gate networks. Through such translation, an AND gate with inputs $\{x, y\}$ and output $z$ becomes $\{[\overline{z}x]$, $[\overline{z}y]$, $[z\overline{xy}]\}$. Similarly, an OR gate is captured by $\{[z\overline{x}]$, $[z\overline{y}]$, $[\overline{z}xy]\}$. About one third of the clauses are ternary in the two most challenging public domain microprocessor verification suites, fvp-sat.3.0 and fvp-unsat.3.0[35]. Furthermore, CDCL tends to generate

a larger number of ternary implicates than binary implicates.

So, it is worthwhile to perform ternary clause BCP through a specialized mechanism. Doing so allows us to favor ternary clause resolutions in CDCL derivations, and also makes BCP faster. The approach we adopt represents each clause with a set of *stub* structures. A stub is a 4-tuple, ($l_a$,$l_b$,$p_a$,$p_b$). Both $l_a$ and $l_b$ are literals, while $p_a$ and $p_b$ are position indexes used to construct antecedents for CDCL. These four fields are packed into 64 bits: 20 bits per literal, 12 bits per index.

Each variable, $v$, is associated with two lists of stubs, $T(v)$ and $T(\overline{v})$. Every stub in $T(\overline{l})$ encodes a binary clause that is implied as a result of literal $l$ being set $true$. Each ternary clause, $[l_0 l_1 l_2]$, is represented by three list entries:

- stub ($l_1$,$l_2$,$p_1$,$p_2$) at position $p_0$ in $T(l_0)$,
- stub ($l_0$,$l_2$,$p_0$,$p_2$) at position $p_1$ in $T(l_1)$, and
- stub ($l_0$,$l_1$,$p_0$,$p_1$) at position $p_2$ in $T(l_2)$.

Position indexes connect together the three stubs that represent a clause. Each stub records the positions of the other two. Before the stubs are inserted, the position indexes are calculated. For all $i \in \{0, 1, 2\}$, position $p_i = |T(l_i)|$. That is, the position index for a list is its cardinality. If $T(l_i)$ contains $n$ stubs, then positions $\{0, 1, ..., n-1\}$ are occupied. The new stub for $T(l_i)$ will be inserted at position $n = |T(l_i)|$.

Suppose a literal, $l$, is set $true$ in BCP step B1. The following procedure is called after binary clause BCP finishes, but before TWL begins. Ternary clause BCP for $l$ occurs in a single pass through $T(\overline{l})$:

**E0.** For each stub $(l_a, l_b, p_a, p_b) \in T(\overline{l})$:

[**E0a.**] If $l_a$ is $true$, skip E0b,...,E0f.

[**E0b.**] If $l_b$ is $true$, skip E0c,...,E0f.

[**E0c.**] If $l_a$ is free, go to E0f.

[**E0d.**] If $l_b$ is $false$, return "falsified $[l_a l_b \overline{l}]$".

[**E0e.**] Enqueue ($l_b$,$p_b$), and skip E0f.

[**E0f.**] If $l_b$ is $false$, enqueue ($l_a$,$p_a$).

Each pair ($l_a$,$l_b$) is a binary clause induced through the falsification of $\overline{l}$. Empirically, most of these clauses have already been satisfied by the time they are considered. It is for this reason that the first two steps, E0a and E0b, check for *true* literals. This speeds BCP, since most clauses are dealt with in one or two comparisons.

When step E0c is reached, each of $l_a$ and $l_b$ are either $false$ or free. Thus, there are four cases to distinguish. If $l_a$ is free, the process enters step E0f, where two of the four cases are handled. Otherwise, execution falls through into step E0d.

In step E0d, $l_a$ is known to be $false$. If $l_b$ is also $false$, there is a conflict, and so falsified clause $[l_a l_b \overline{l}]$ is returned explicitly. Else, $l_b$ is free and implied; it is inserted into the BCP queue with position index $p_b$ as its antecedent. CDCL interprets this antecedent as a reference to the stub at position $p_b$ in $T(l_b)$. This stub contains $l_a$ and $\overline{l}$, sufficient antecedent information to support a CDCL resolution on $l_b$. To ease correct interpretation, antecedent information of this variety must be labeled to distinguish it from the pointer antecedents of TWL and the literal antecedents of binary BCP.

If step E0f is reached, $l_a$ is free. If $l_b$ is $false$, $l_a$ is queued for assertion. Else, $l_b$ is free, and so the current ternary clause has no BCP consequence.

In TWL, only two of the literals in any ternary clause are watched. There is always one literal without a link to the clause. In the above scheme, all three literals have a link. But, the scheme is so efficient that this weakness is more than compensated for. It is efficient for the same reasons the mechanism of section 3.5.1 is efficient. Compared to ternary BCP via TWL, fewer instructions are executed per clause. Use of a bounds-test sentinel in each stub list provides further savings. A contiguous representation of $T(\overline{l})$ fits as many as four stubs on a 32-byte cache line. Sequential traversal makes good use of L1, and relative to TWL, fewer lines are pulled into cache.

Having position indexes 12 bits wide implies that a ternary clause, $[l_0 l_1 l_2]$, cannot be represented with stubs if $|T(l_i)| = 4096$, for some $i \in \{0, 1, 2\}$. Instead, the general purpose TWL representation is used. However, in our experience, it is rare for any one literal to be present in 100 ternary clauses, let alone 4096.

### 3.5.3 Clause Compression

In a ternary clause stub, each literal is allocated 20 bits. Setting aside one bit for the sign leaves 19 bits for the variable index. Therefore, the maximum index that may be encoded is $2^{19} - 1$. As a result, the solver supports no more than 524,288 variables. This restriction

reduces the set of instances to which the solver is applicable. But, to put this in perspective, we know of just one benchmark suite with formulas having $> 100,000$ variables[24]. (Although these formulas are large, they are not difficult, and Chaff solves them quickly. The benchmark authors state the formula sizes are a product of inefficient encoding.) Most interesting benchmarks contain $< 40,000$ variables. It is reasonable to sacrifice the capacity to solve abnormal, gargantuan formulas if doing so allows us to better solve the formulas we expect will normally arise.

The TWL procedure we have described operates on clauses that are represented by arrays of literal instances. A literal instance consists of a 20-bit literal, paired with a 1-bit watched flag. Clearly, it is possible to pack three 21-bit literal instances into 64 bits of memory. It is straightforward to arrange fields so that unpacking is inexpensive. One such arrangement allows two instances to each be isolated in a single bitwise operation, while the third instance is available in four bitwise operations on a 32-bit machine (two on a 64-bit machine). In contrast, Chaff maintains a 32-bit structure for each literal occurrence.

A more compact representation decreases the width in memory of each clause. Storing three literals, instead of two, in each 64-bit block allows 12 literals, instead of 8, per 32-byte cache line. So, fewer cache lines are touched in the course of TWL clause traversals. Short searches, which are very common for TWL, complete more often within a single cache line. Long searches have lessened impact. Because of TWL's relaxed watch placement rules (versus HTL), an assignment is deduced only after full traversal of its antecedent clause. The reader is referred back to Table 2.3, in which some average first-UIP clause lengths are listed. Using 21 bits instead of 32, a clause with 100 literals occupies 4 fewer cache lines. A clause with 600 literals occupies 25 fewer. Less main memory is copied into cache, so latency costs and cache pollution are reduced.

### 3.5.4 Receding Boundary Sentinels

The TWL procedure we have presented takes advantage of boundary sentinels to improve the efficiency of the W0b/W0c search loop. These sentinels are literals that evaluate *true* or free. Rather than testing at every literal whether the search pointer has been directed outside the clause array, this test occurs only when a non-$false$ literal is encountered. Because the search loop is so simple, the removal of a comparison is significant (cf. Knuth's discussion of linear search[23]). Boundary sentinels also contribute to another refinement we have developed.

The technique excludes some $false$ literal instances from being considered during BCP. If a literal is made $false$ at decision level $y$, it will remain $false$ until dl $y$ is removed through backtracking. Empirically, less than four decision levels, on average, are removed by a backtrack—typically, only a small fraction of the current branch. DLL lingers in the fringe of its tree, even with nonchronological backtracking. Literal instances become $false$ and then remain that way through long periods of search.

This motivates the omission of $false$ literals from clauses in which they occur. Clearly, if TWL works with fewer literals, BCP will execute faster. It is less apparent that such literals can be removed efficiently. Already, TWL handles $false$ literals very fast. The gain must outweigh the cost if the technique is to be advantageous.

In fact, it is possible to remove numerous $false$ literals from the formula at minor expense. A conflict occurs when a clause, $F_X$, is falsified. In response, CDCL introduces an implicate clause, $F_I$, that is also entirely $false$. Before backtracking to make $F_I$ unit, we consider truncating both $F_X$ and $F_I$.

For each $c \in \{F_X, F_I\}$, the following procedure is applied. If $c$ is less than some threshold length, it is not truncated. This helps to avoid situations where the improvement to BCP is offset by the truncation cost. Otherwise, the literal instances within $c$ are sorted according to decision level. To clarify, let $c$ be $[l_1 l_2 ... l_n]$. Suppose each literal $x \in c$ became $false$ at decision level $d(x)$. The contents of the array representing $c$ are rearranged so that, for all $l_i, l_j \in c$: if $d(l_i) < d(l_j)$, the memory address of $l_i$ is less than that of $l_j$. Bentley-McIlroy 3-way quicksort[31] is recommended for this task, since there are often many duplicate keys; i.e., it is typical for several literals in $c$ to have been set $false$ together, at the same dl.

Once the array is sorted, the two literal instances highest in memory become watched. Because the literals in $c$ have been sorted by dl, the last literal is at the shallowest dl, while its neighbor is either at the shallowest (if $c = F_X$) or second-shallowest (if $c = F_I$). Therefore, backtracking will not violate the TWL requirement.

Once the clause is watched, boundary sentinels are placed. One of the two sentinels, $S_H$, occupies the highest memory location in the array. The other sentinel, $S_L$, is initially positioned some small distance lower in memory than $S_H$. The two watched literals for $c$ lie between $S_L$ and $S_H$. $S_L$ serves to partition the array into two subarrays, $A_L$ and $A_H$.

Subarray $A_H$ is bounded on either side by $S_L$ and $S_H$. $A_L$ is the complement of $A_H$, i.e., $A_L = c - A_H$. Because both watches are embedded in $A_H$, the contents of $A_L$ are ignored by TWL. This improves the efficiency of BCP. During backtracking, $S_L$ is moved through $c$

to ensure that every literal in $A_L$ is $false$. But, $S_L$ is only moved lower in memory, never higher; backtracking widens $A_H$ and narrows $A_L$.

Let $d(A_L)$ be the label of the shallowest dl at which some literal in $A_L$ became $false$. Because $A_L$ is sorted by decision level, $d(A_L)$ is the dl of the literal in $A_L$ with the highest memory address. Unless $A_L$ is empty, the solver stores a link from decision level $d(A_L)$ to sentinel $S_L$. Such links allow sentinels to be moved efficiently in response to backtracking. Whenever backtracking removes a dl, $d$, the solver moves all sentinels to which $d$ is linked. Then, some or all of these sentinels are relinked, and all of $d$'s links are discarded.

When backtracking removes decision level $d(A_L)$, $S_L$ is moved. If possible, this narrows $A_L$ to $A'_L$, such that $A'_L$ is not empty and $d(A_L) > d(A'_L)$. If no such $A'_L$ exists, $S_L$ is put to rest at the lowest memory location in the array. Otherwise, decision level $d(A'_L)$ is linked to $S_L$, and all links for decision level $d(A_L)$ are deleted.

To reiterate, boundary sentinels are used to efficiently truncate long, falsified clauses. Whenever a $false$ clause of sufficient length is available, the solver performs a truncation. Suppose $c$ is such a clause. First, the literals of $c$ are arranged in order of increasing decision level. Second, the two literals ranked highest in this ordering become watched. This is necessary because sorting $c$ may invalidate watch pointers into $c$. Third, a boundary sentinel is placed at the high end of the clause. This sentinel need not be moved. Fourth, another boundary sentinel is used to truncate $c$. Instead of being placed at the low end of the clause, it is placed midway through, dividing $c$ into $A_H$ and $A_L$. $A_H$ is the shortened clause; $A_L$ is an array of omitted $false$ literals.

Shortened clauses are lengthened to include omitted literals that become free. $A_H$ must be extended if a literal in $A_L$ becomes free. A clause is lengthened by moving its low memory sentinel during backtracking. To facilitate this, each decision level, $d$, is linked to every sentinel that must be repositioned when $d$ is removed. When backtracking frees a literal in $A_L$, all literals in $A_H$ are already free. $A_H$ must be expanded to include the free literals in $A_L$. So, the low memory sentinel is moved to a lower address. If all literals in $A_L$ are free, the sentinel is moved off the low end of the clause—$A_H$ expands to contain the whole of $c$. Otherwise, a contiguous block of literals from the high end of $A_L$ is transferred into $A_H$, and the sentinel becomes linked to the dl of the highest memory literal in the remainder of $A_L$.

# Chapter 4

# Decision Strategy

## 4.1 Overview

The focus of Chapter 3 is efficient implementation of the procedures executed during search. Efficient BCP is important because, all else being equal, a gain in BCP speed produces a proportional gain in solver speed overall. But, what is most important is the solver's capacity to find and take advantage of useful problem structure when it is available. Brute force combinatorial search rapidly becomes futile as problem size increases, no matter how efficiently it is implemented. The better the solver can exploit structural simplicity, the more the difficulty of an instance can depend on its complexity rather than on its size. Ideally, if a formula has a short resolution refutation, superficial characteristics, e.g., the number of variables, should be immaterial.

Effective decision methods discover useful structural properties of the formula, and guide search to exploit them. Good DLL decision strategies work to restrict search space, in order to facilitate exhaustive search[19]. They guide the solver to find tree-like refutations that involve fewer resolutions. In contrast, good strategies for clause-learning DLL solvers work to generate clusters of compositionally similar, resolvable clauses. Conflict-driven clause learning takes advantage of these clusters, since it tends to resolve together clauses that share literals.

First, we present a brief overview of the best general purpose decision strategies for DLL. Then, we describe the VSIDS decision strategy used in Chaff, and explain why it works. On that foundation, we introduce a new and superior decision strategy. Finally, we discuss Berkmin's contribution.

## 4.2 Strategies for DLL

Heuristics may be designed to suit particular classes of instances. For example, SATO employs a specialized strategy to solve quasigroup problems[40]. Here, we consider general purpose strategies only.

In DLL, the decision strategy selects a variable, $v$, to branch on. It also dictates the order in which the literals of $v$ are asserted, although this is relatively unimportant. The decision strategy determines the search tree.

The strategies that seem to be most successful at producing desirable trees are guided by formula simplification. Formula $F_1$ is said to be "simpler" than formula $F_2$ if $g(F_1) > g(F_2)$, for $g$ defined as follows. Function $g$ computes an exponentially weighted sum of the clause sizes for a given formula. That is, $g(F) = \sum_{c \in F} k^{-s(c)}$, where $F$ is a CNF from which all $false$ literals and satisfied clauses have been removed; $k$ is some experimentally determined constant; and $s$ is a function that returns the number of literals in a given clause. A clause of size $x$ contributes as much to the sum as $k$ clauses of size $x + 1$.

The best DLL solvers, e.g., Satz[25] and POSIT[14], work to reduce the number of decisions along each path from the root to a leaf. At each branching point in the tree, an attempt is made to select a variable to minimize the number of decisions needed to complete the tree rooted at that point. This is an efficient approach because most (or all, if the formula is unsatisfiable) induced formulas will be refuted[19]. Variables are ranked by their power to simplify the formula. It is assumed, but not proved, that a larger $g(F)$ implies a smaller expected number of decisions to refute $F$. The intuitive justification that appears in the literature is along the lines of: minimizing the number of free variables minimizes a loose bound on the maximum tree size; a formula with more short clauses is more constrained, and therefore closer to a conflict; and so on. The strong support is empirical.

Suppose a decision is needed for formula $F$. Let $F|_x$ denote the formula produced by BCP, given the input $F \wedge [x]$. An elementary decision strategy that is guided by formula simplification chooses to branch on a variable, $v$, that maximizes $g(F|_v) * g(F|_{\overline{v}})$. More sophisticated methods reduce the time spent making decisions by ignoring long clauses in the computation of $g$, pruning the set of variables for which $g$ is computed, etc.

Every DLL search tree is linearly equivalent to a tree-like resolution refutation[15]. Each decision corresponds to the resolution of two clauses, both of which were derived through one or more resolutions. (Unless the decision variable did not play a role in one of the

subtree refutations.) But, each BCP assignment corresponds to a resolution involving a clause from the input formula. Assertions that heavily simplify the formula are typically those that generate numerous consequential assignments through BCP. And, short clauses promote BCP. Therefore, simplification heuristics are essentially geared toward minimizing the number of resolutions in the refutation.

## 4.3 Strategies for DLL with CDCL

Relsat's decision strategy[3] is closely related to the formula simplification heuristics used in Satz and POSIT. The same is true of SATO's general purpose strategy[40]. In GRASP, a variety of decision strategies are implemented. Several of them are just DLL formula simplification heuristics. However, others are *literal count* (LC) heuristics. In fact, GRASP's default decision strategy is an LC heuristic[29].

The literal count heuristics introduced in GRASP rank variables according to the number of times they appear in unsatisfied clauses. For example, the *dynamic largest individual sum* (DLIS) strategy counts the number of unsatisfied clauses each literal occurs in. The literal with the largest number of occurrences is set *true*. Experimental results indicate that within the GRASP framework, DLIS does better than the best DLL formula simplification heuristics on non-random formulas[27]. The published explanation for this result is that (a) classical DLL strategies are too greedy, and (b) decision strategy is not important for a clause-learning solver. The first claim is dubious, given the results in, e.g., [25]: greedier heuristics seem to be preferable, except they are too expensive to compute. The second claim is refuted by VSIDS.

### 4.3.1 VSIDS

Arguably, Chaff's most important contribution is the *variable state independent decaying sum* (VSIDS) decision strategy[30]. VSIDS is a literal count heuristic that is dramatically more powerful than DLIS. It allows Chaff to solve difficult industrial SAT problems much faster, with far fewer decisions, than solvers like Relsat, GRASP, and SATO.

VSIDS is realized in zChaff as follows. Each literal, $l$, has a score, $s(l)$, and an occurrence count, $r(l)$. When a decision is necessary, a free literal with the highest score is set *true*. Initially, for every literal, $l$, $s(l) = r(l) = 0$. Before search begins, $s(l)$ is incremented for each occurrence of a literal, $l$, in the input formula. When a clause, $c$, is learned during

search, $r(l)$ is incremented for each literal $l \in c$. Every 255 decisions, the scores are updated: for each literal, $l$, $s(l)$ becomes $r(l) + s(l)/2$, and $r(l)$ becomes zero.

VSIDS deviates from DLIS in two respects. First, literal occurrences within satisfied clauses are not distinguished. As discussed in Section 3.5.4, search lingers in the fringe of the tree; that is where most decisions are made. Normally, in the fringe, most variables have been assigned a value, and many clauses are satisfied. Counting occurrences in satisfied clauses makes a substantial difference in the literal ranking. Still, the published explanation for why VSIDS is successful begins with the claim that VSIDS works to satisfy conflict clauses. DLIS is described as working to satisfy clauses, but in a myopic way, ignoring the impact of BCP in order to avoid excessive greed. VSIDS is described as a myopic, inaccurate attempt at satisfying clauses (especially those recently derived). The explanation for VSIDS is even less convincing than the explanation for DLIS.

Second, the influence of each occurrence is scaled according to the occurrence's recency. Decisions are based on scores, not occurrence counts. As clauses are learned, occurrence counts are incremented. Periodically, the scores are halved, the occurrence counts are added to the scores, and the occurrence counts are zeroed. These updates are frequent: it depends on the instance, but typically, 255 decisions translates to about 64 conflicts. It is the emphasis on recent literal occurrences that is the crux of VSIDS' power. The published explanation is that, on difficult problems, conflict clauses drive the search process; therefore, favoring the information in recent clauses is valuable. Clearly, this explains nothing.

In our view it is a mistake to cast literal count heuristics as sloppy approximations to DLL formula simplification heuristics. Instead, we propose VSIDS is actually a clause learning heuristic that guides the solver to generate clusters of related, resolvable clauses.

At each leaf in the search tree, CDCL resolves clauses that were involved in BCP along the path to that leaf. Suppose CDCL derives two implicates, $i_0$ and $i_1$. Suppose both derivations occur with many of the same literals asserted. Then, it is likely there are a substantial number of literals that participate in both derivations. As a result, it tends to be that $i_0$ and $i_1$ are compositionally similar. That is, they tend to contain the same literals. We have confirmed this empirically.

Nonchronological backtracking typically removes only a small fraction of the path from a leaf to the root. It is usual for most of the path to remain intact from one conflict to the next. Therefore, from one conflict to the next, many of the same literals remain asserted. Implicates produced in leaves that share a long path prefix (leaves that are, in the obvious

sense, nearby) are often compositionally similar.

VSIDS' exponential decay of variable scores shifts focus toward recently derived clauses, i.e., those that tend to have been learned near the current search position in the DLL tree. VSIDS selects a free literal, $l$, that has the highest score, and makes it $true$. Since CDCL implicates consist of $false$ literals, this fosters the conflict-driven derivation of clauses that contain $\overline{l}$. Thus, VSIDS guides the solver to generate implicates resolvable against clauses that are usually of similar composition.

### 4.3.2 VMTF

There are at least two problems with VSIDS as a method of learning related, resolvable clauses. First, periodic score decay is an indirect and awkward means of choosing decision literals from recently derived clauses. There is a delay in the use of gathered statistics, so focus does not shift immediately. Until the literal scores are updated, the most recent clauses are ignored. Ironically, because of the depth-first organization of DLL search, these are the clauses produced in the leaves that are most likely to share a long path prefix with the current search position in the DLL tree.

Second, if a pair of clauses clash on more than one variable, they cannot be resolved by conflict-driven learning; a resolvent would be tautologous. For example, if $[xyP]$ is resolved against $[\overline{xy}Q]$, the resolvent must contain both $x$ and $\overline{x}$, or both $y$ and $\overline{y}$. Suppose two literals, $l_0$ and $l_1$, in a clause, $c_x$, are set $true$ along the path to a leaf in which a clause, $c_d$, is derived. It may happen that $c_d$ contains both $\overline{l_0}$ and $\overline{l_1}$, in which case $c_x$ and $c_d$ are not resolvable. So, the solver should perhaps tend to avoid setting more than one literal $true$ in any of the recent clauses.

Primarily in response to the first problem, we introduce the *variable move-to-front* (VMTF) decision strategy. VMTF is simple and extremely inexpensive to compute. More importantly, if our solver uses VMTF instead of VSIDS, far fewer decisions are needed to solve benchmarks from various interesting domains: planning, bounded model checking, circuit equivalence checking, and so on, as shown in Table 4.1.

An occurrence count, $r(l)$, is kept for each literal, $l$. Initially, for every $l$, $r(l) = 0$. Before search begins, $r(l)$ is incremented for each occurrence of a literal, $l$, in the input formula. An ordered list of the formula variables, $W$, is also maintained. Once the counts have been determined for the input formula, $W$ is arranged so that, for all variables $v_0$, $v_1$: if $r(v_0) + r(\overline{v_0}) > r(v_1) + r(\overline{v_1})$, then $v_0$ precedes $v_1$. The more often a variable occurs in

the input formula, the closer it begins to the front of the list.

When a clause, $c$, is learned during search, $r(l)$ is incremented for each literal $l \in c$. Then, some of the variables in $c$ are moved to the front of $W$. The number of variables moved is a small constant, $m$, e.g., 8. If $c$ contains only $n < m$ literals, $n$ variables are moved. The moved variables are positioned at the front of the list in an arbitrary order. When a decision is necessary, the free variable, $v$, that is nearest the front of $W$ is set $true$ if $r(v) > r(\overline{v})$, $false$ if $r(\overline{v}) > r(v)$, and randomly otherwise.

Recall that once a clause, $c$, is derived, the solver backtracks to the deepest decision level at which $c$ is unit. At that point, BCP satisfies $c$; every variable in the clause is assigned. Therefore, none of the variables in $c$ that are moved to the front of $W$ are free when the next decision is made. Clearly then, VMTF does not simply choose variables from the most recently derived clause.

Because implicates that are learned near each other in the DLL tree tend to share literals, it is often the case that many of the variables moved to the front are already near the front, prior to being moved. But, if every variable is moved to the front for every learned clause, performance degrades substantially, relative to VMTF as described above. That is, a larger number of decisions are needed to solve the same instances. Moving only a few variables from each clause prevents a single clause having too large an impact on the decision making process.

It suffices to move to the front of $W$ a random selection of $m$ variables from $c$. However, a more systematic approach leads to better results. It is not beneficial to favor moving the variables at the shallowest decision levels, i.e., the variables that will become free earliest. Rather, it is better to move the variables from $c$ that appear earliest in the participation trace for the derivation of $c$. The reasons for this are unclear.

Further gains are possible if a broader set of variables is considered while making each decision. One approach that works well is choosing between the first two free variables

| **instance** | **VMTF** | **VSIDS** | **instance** | **VMTF** | **VSIDS** |
|---:|:---:|:---:|---:|:---:|:---:|
| 3pipe[37] | 32,121 | 48,443 | longmult15[6] | 151,921 | 112,790 |
| 6pipe[37] | 1,716,486 | 5,174,285 | barrel9[6] | 128,041 | 571,707 |
| 7pipe[37] | 4,788,532 | >9,999,999 | 9vliw_bp_mc[37] | 1,415,636 | 2,918,058 |
| 7pipe_bug[37] | 574,158 | >9,999,999 | hanoi6[34] | 567,050 | >9,999,999 |

Table 4.1: Number of decisions used to complete proof

in $W$ using a scoring scheme reminiscent of VSIDS. Each variable, $v$, has a score, $s(v)$. Initially, the score for each variable is zero. When a clause, $c$, is learned during search, $s(v)$ is incremented for each variable, $v$, that has a literal in $c$. Periodically, all the scores are divided by a constant that is a power of two.

When a decision is needed, the following procedure is used. Let $v_0$ and $v_1$ be the first two free variables in $W$. Assume $v_0$ precedes $v_1$, and the number of list elements between $v_0$ and $v_1$ in $W$ is $d$. If the score for $v_1$ exceeds the score for $v_0$ by wide enough a margin, $v_1$ is selected instead of $v_0$. The larger $d$ is, the wider the margin required. For example, if $s(v_0) + 2d + 3 > s(v_1)$, then $v_0$ is selected, else $v_1$ is selected.

Finally, we note that VSIDS is very inexpensive to compute, compared to the most effective formula simplification heuristics, and that VMTF is much cheaper to compute than VSIDS. The VSIDS implementation in zChaff accounts for about ten percent of the runtime. Most of that time is spent sorting literals by score. Our VMTF implementation accounts for less than one percent of our solver's runtime.

### 4.3.3 Berkmin

One decision strategy that addresses both of the problems with VSIDS is the heuristic introduced in [16], a paper that was published subsequent to our development of VMTF. We neglect to include here a full and detailed description of the heuristic, since it is complex and the cited source is sufficiently lucid.

The strategy is essentially as follows. Each variable has a score that is initially zero. Each time a clause participates in the derivation of an implicate, the score for every variable in the clause is incremented. Periodically, the scores are divided, as in VSIDS.

When a decision is necessary, the strategy considers $c$, the most recently derived clause that is yet unsatisfied. If there is no such $c$, because there are no derived clauses, or because all derived clauses are satisfied, a variable with the highest score overall is selected. Whether this variable is set $true$ or $false$ depends on an estimate of which choice will generate more assignments through BCP.

Otherwise, the heuristic selects $v$, one of the variables in $c$ that has the highest score among all variables in $c$. The literal of $v$ that has occurred in the largest number of implicates is set $true$. If both literals have appeared equally often, a random choice is made.

The published explanation for why this works is unconvincing, along the lines of: it is like VSIDS, only more dynamic. But, our explanation for literal count strategies predicts

Berkmin's heuristic will be successful. It is clearly similar to VMTF.

We have experimentally verified several relevant facts. First, in general, sign selection guided by counting BCP assignments is not beneficial. It suffices to make *true* the literal that has occurred in the largest number of implicates.

Second, it is not important that decision variables be selected from unsatisfied clauses. Rather, it is important that decision variables be selected from clauses that contain no more than one *true* literal. For example, a variable may be drawn from a clause that already contains exactly one *true* literal, unless doing so will produce a second *true* literal in the clause. This works better than selecting from unsatisfied clauses, consistent with the notion that the decision strategy operates to promote production of clauses that can be resolved against recent implicates.

# Chapter 5

# Related Work

The head/tail lists BCP algorithm was introduced in SATO[40]. The two watched literals BCP algorithm was introduced in Chaff[30], and its memory access properties were noted.

Iterative DLL with clause learning and nonchronological backtracking was introduced in [29] and improved in [3]. Conflict-driven clause learning was first used in the solvers GRASP[29] and Relsat[3]. Chaff's first-UIP learning scheme was introduced in [41]. A very thorough experimental evaluation of several learning schemes appears in [41].

The DLIS decision strategy was introduced in [29], and was experimentally compared against DLL formula simplification heuristics in [27]. The VSIDS decision strategy was introduced in [30]. Extensive improvements to VSIDS are published in [16].

# Chapter 6

# Conclusions

Many interesting problems are solved efficiently in practice through translation to SAT. This is largely because modern satisfiability solvers are frequently able to derive and take advantage of simple structures in problem instances. Although the heuristics these solvers apply are crude and indirect, they are remarkably successful. We believe there is potential for enormous improvement.

Boolean constraint propagation is used both to select clauses for resolution, and to prune space during search for a satisfying assignment. BCP is slow because it operates diffusely over a data structure that is much larger than the cache. We emphasize that the most significant performance gains are achieved by reducing the number of accesses to main memory. We have presented a very refined two-pointer BCP algorithm that is simpler and more efficient than the one found in Chaff. We have proposed a pair of new binary and ternary clause BCP algorithms that have ideal memory access properties. Recognizing that a solver designed to handle a large number of variables should be quite different than a solver designed to handle fewer variables, we have used packed representations to improve BCP locality. We have developed our usage of clause boundary sentinels into a straightforward and effective method of excluding false literals from consideration during BCP. Using these improvements, our solver is dramatically faster than, for example, Chaff.

Conflict-driven clause learning is the most important difference between modern solvers and DPLL solvers like POSIT and Satz. CDCL is usually discussed in terms of cuts through literal implication graphs. For example, see [41] and [4]. We view this as a misleading approach that obscures the essence of the technique, and have instead presented CDCL in terms of resolution. This has facilitated our simple and complete coverage of the algorithm,

including several clear proofs. We have illustrated that CDCL operates as a resolution heuristic, and we have begun to explain why some learning schemes are better than others. We suggest for future work a separation between clause learning and backtracking.

The success of decision strategies like VSIDS cannot be explained by an appeal to formula simplification arguments. We realize this and have argued that literal count decision heuristics are actually a means of guiding the solver to learn clauses that are both resolvable and compositionally similar. Working with this hypothesis we have developed a new decision heuristic that allows our solver to perform extremely well over a wide range of problem classes.

# Bibliography


[1] F. Aloul and K. Sakallah, "An experimental evaluation of conflict diagnosis and recursive learning in boolean satisfiability," in *Proceedings of the International Workshop on Logic Synthesis (IWLS)*, pp. 117–122, 2000.

[2] F. Aloul, B. Sierawski, and K. Sakallah, "Satometer: How much have we searched?," in *Proceedings of the 39th Design Automation Conference (DAC'02)*, pp. 737–742, 2002.

[3] R. J. J. Bayardo and R. C. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, (Providence, Rhode Island), pp. 203–208, 1997.

[4] P. Beame, H. Kautz, and A. Sabharwal, "Understanding the power of clause learning," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, (Acapulco, Mexico), 2003.

[5] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson, "Optimal seperation of treelike and general resolution." To appear in Combinatorica.

[6] A. Biere, A. Cimatti, E. 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), number 1579 in LNCS*, 1999.

[7] P. Chong and M. Prasad, "Why is atpg easy?," in *Proceedings of the 36th Design Automation Conference (DAC '99)*, pp. 22–28, June 1999.

[8] V. Chvtal and E. Szemerdi, "Many hard examples for resolution," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 759–768, 1988.

[9] S. A. Cook and D. G. Mitchell, "Finding hard instances of the satisfiability problem: A survey," in *Satisfiability Problem: Theory and Applications* (Du, Gu, and Pardalos, eds.), vol. 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pp. 1–17, American Mathematical Society, 1997.

[10] J. Crawford and D. Wang, "International competition and sympoisium on satisfiability testing," March 1996. http://www.cirl.uoregon.edu/crawford/beijing/.

[11] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," in *Communications of the ACM*, vol. 5, pp. 394–397, 1962.

[12] M. Davis and H. Putnam, "A computing procedure for quantification theory," in *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[13] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-SAT formulae," in *IJCAI*, pp. 248–253, 2001.

[14] J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, Departement of computer and Information science, University of Pennsylvania, Philadelphia, 1995.

[15] A. V. Gelder, "Combining preorder and postorder resolution in a satisfiability solver," in *Electronic Notes in Discrete Mathematics* (H. Kautz and B. Selman, eds.), vol. 9, Elsevier, 2001.

[16] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Design, Automation, and Test in Europe (DATE '02)*, pp. 142–149, Mar. 2002.

[17] P. L. Hammer and S. Rudeanu, *Boolean Methods in Operations Research and Related Areas.* Springer-Verlag, Berlin, Heidelberg, New York, 1968.

[18] E. Hirsch and A. Kojevnikov, "UnitWalk: A new SAT solver that uses local search guided by unit clause elimination," 2001. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg.

[19] J. N. Hooker and V. Vinay, "Branching rules for satisfiability," *Journal of Automated Reasoning*, vol. 15, pp. 359–383, 1995.

[20] *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, (Nagoya, Japan), August 23–29 1997.

[21] Intel, *Pentium 4 Developer's Guide*. 2000.

[22] H. A. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pp. 359–363, 1992.

[23] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading MA, second ed., 1998.

[24] T. Leonard, "Bounded model checking an alpha design." http://www.ftp.cl.cam.ac.uk/ftp/hvg/sat-examples/.

[25] C.-M. Li and Anbulagan, "Heuristics based on unit propagation for satisfiability problems," in IJCAI97 [20], pp. 366–371.

[26] I. Lynce and J. 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)*, December 2001.

[27] J. P. Marques-Silva, "The Impact of Branching Heuristics in Propositional Satisfiability Algorithms," in *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, September 1999.

[28] J. P. Marques-Silva and T. Glass, "Combinational Equivalence Checking Using Satisfiability and Recursive Learning," in *Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, 1999.

[29] J. P. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 220–227, November 1996.

[30] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.

[31] R. Sedgewick, *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., 1992.

[32] M. Sheeran and G. Stålmarck, "A tutorial on Stålmarck's proof procedure for propositional logic," in *Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998* (G. Gopalakrishnan and P. Windley, eds.), vol. 1522, pp. 82–99, Berlin: Springer-Verlag, 1998.

[33] L. Simon and D. L. Berre, "The SAT2003 competition." http://www.satlive.org/SATCompetition/2003/index.jsp.

[34] L. Simon, D. L. Berre, and E. A. Hirsch, "The SAT2002 competition." http://www.satlive.org/SATCompetition/2002/index.jsp.

[35] M. Velev, "Using rewriting rules and positive equality to formally verify wide-issue out-of-order microprocessors with a reorder buffer," in *Design, Automation and Test in Europe (DATE '02)*, pp. 28–35, March 2002.

[36] M. Velev and R. Bryant, "Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic," in *Correct Hardware Design and Verification Methods (CHARME '99)*, pp. 37–53, September 1999.

[37] M. Velev and R. Bryant, "Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors," in *38th Design Automation Conference (DAC '01)*, pp. 226–231, June 2001.

[38] E. Zarpas, "IBM formal verification benchmarks library." http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/benchmarks.html.

[39] 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.

[40] H. Zhang, "SATO: an efficient propositional prover," in *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pp. 272–275, 1997.

[41] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik, "Efficient conflict driven learning in a Boolean satisfiability solver," in *International Conference on Computer-Aided Design (ICCAD'01)*, pp. 279–285, Nov. 2001.