Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Joonas Lipping

Dense Implicant Cubes in Boolean Satisfiability Problems

Master's Thesis
Espoo, December 18, 2019

Supervisor: Adjunct Professor Tomi Janhunen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author: Joonas Lipping

Title:
Dense Implicant Cubes in Boolean Satisfiability Problems

Date: December 18, 2019 **Pages:** 66

Major: Computer Science **Code:** SCI3042

Supervisor: Adjunct Professor Tomi Janhunen

The Boolean satisfiability problem (SAT) is to determine whether a formula of propositional logic evaluates to true under some assignment of truth values to its variables. It provides a natural formalism for a wide range of computational tasks, such as planning and verification tasks. SAT is NP-complete in the general case, meaning that no known solver can efficiently solve arbitrary SAT instances. However, the instances that emerge from real-world applications often have plentiful internal structure, which modern solvers can exploit using various inference techniques, making them tractable in practice. The effectiveness of such techniques has a pivotal impact on solving performance.

This thesis presents a density measure for scoring certain conjunctions of literals based on the number of implied assignments they yield under unit propagation, and a solving technique that uses this score as a heuristic for the expected refutation work. An algorithm than incorporates the technique into a CDCL-type solver is described and implemented as an extension to the MiniSat 2.1 solver. The modified solver is evaluated by comparing its running times on a set of benchmark instances to those of the original solver, using both the original-form benchmarks and their permutations. The modified solver is slower on average, solving 35% of original-form and 29% of permuted-form instances faster than the baseline. The evidence, therefore, does not support using this algorithm in general. However, the modified solver appears to have the advantage on the satisfiable instances of particular benchmark families (in their non-permuted form), indicating that the algorithm may be of some use for specific kinds of instances.

Keywords: Boolean satisfiability, propositional logic, propositional satisfiability, SAT solver

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Joonas Lipping		
Työn nimi:	Tiheät implikanttikuutiot Boolen toteutuvuusongelmissa		
Päiväys:	18. joulukuuta 2019	**Sivumäärä:**	66
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Adjunct Professor Tomi Janhunen		

Boolen toteutuvuusongelmana (SAT) on selvittää, voidaanko propositiologiikan lauseen muuttujien totuusarvot valita siten, että lause on tosi. SAT on luonnollinen tapa ilmaista monia laskennallisia tehtäviä, muun muassa suunnitteluja verifiointitehtäviä. Ongelma on yleisessä muodossaan NP-täydellinen, eikä sellaista ratkaisumenetelmää tunneta, joka kykenisi ratkaisemaan mielivaltaisia SAT-instansseja tehokkaasti. Monissa käytännön sovelluksissa vastaan tulevissa instansseissa on kuitenkin runsaasti sisäistä rakennetta, jota nykyratkaisimet kykenevät hyödyntämään erinäisin tekniikoin. Tällä tavoin instanssit ovat käytännössä ratkaistavissa. Tunnettujen tekniikkojen toimivuudella on määräävä merkitys ratkaisinten suorituskyvyssä.

Tässä työssä esitellään tiheysmitta, jolla pisteytetään literaalien konjunktioita sen perusteella, kuinka monta muuttujan totuusarvoa niistä seuraa yksikköpropagaatiota käyttämällä, sekä ratkaisutekniikka, joka käyttää tätä pisteytystä heuristiikkana odotetulle refutaatiotyölle. Kehitetään algoritmi, joka sisällyttää tekniikan CDCL-tyyppiseen ratkaisimeen, ja toteutetaan se MiniSat 2.1-ratkaisimen laajennuksena. Muutettua ratkaisinta arvioidaan vertaamalla sen ratkaisuaikoja alkuperäisen ratkaisimen aikoihin, kun ratkaistavana on joukko teollisia vertailuinstansseja sekä niiden sekoitettuja muotoja. Muutettu ratkaisin on keskimäärin hitaampi: se ratkaisee verrokkia nopeammin 35% alkuperäisessä muodossa olevista ja 29% sekoitetuista instansseista. Tämän perusteella algoritmia ei siis voi suositella yleiseen käyttöön. Muutettu ratkaisin näyttää kuitenkin pärjäävän verrokkia paremmin joidenkin instanssiperheiden toteutuvilla instansseilla (ei-sekoitetussa muodossa), joten algoritmista saattaisi olla hyötyä joidenkin tiettyjen instanssityyppien ratkaisemisessa.

Asiasanat:	Boolen toteutuvuus, lauselogiikka, propositiologiikka, toteutuvuustarkastin
Kieli:	Englanti

Contents

Chapter 1

Introduction

The Boolean satisfiability problem, commonly abbreviated as SAT, is to determine whether a given formula of propositional logic evaluates to true under some possible assignment of truth values to its variables. This problem is NP-complete and is, in fact, the first problem for which this property was proven [15]. Therefore, no known algorithm can solve it efficiently in the general case. SAT instances of practical interest to industry, however, often have a lot of internal structure which can be exploited to improve solver performance. Thanks to a great deal of research and engineering effort, modern SAT solvers are quite good at leveraging this internal structure, as well as using various other optimization techniques, often making even quite large problem instances tractable in practice. Nevertheless, they are still fundamentally based on a search in a space with size exponential in the number of problem variables.

The sophistication of modern solvers makes SAT an attractive choice of formalism for many applications. It is often also a natural choice, since propositional logic is well suited for expressing systems of rules such as model descriptions, specifications, and constraints, and the widely useful task of demonstrating the tautologyhood of a propositional formula is equivalent to demonstrating the unsatisfiability of its negation. In science and in industry, SAT solvers are used for many different tasks, such as design and planning, formal verification, and automated theorem proving.

For industrial SAT instances, solvers with a so-called conflict-driven clause learning (CDCL) structure are typically the most effective. Such solvers are the modern evolution of the rather straightforward Davis-Putnam-Logemann-Loveland (DPLL) search procedure, introduced in 1962 [17] and since augmented with various performance techniques, most prominently with heuristics that guide the search and with mechanisms to explicitly record new constraints that the procedure implicitly derives. The technique presented and

evaluated in this work is implemented as a modification of the CDCL-type solver MiniSat 2.1.

1.1 Contribution

The operation of a SAT solver can be understood as a search in the space of variable assignments. This work proposes a heuristic for the relative solvability of certain types of regions (namely, cubes of assignments) of this search space, a principle for finding regions where this heuristic is favorable, and an algorithm that incorporates this principle into a CDCL solver.[1] If the heuristic is sufficiently descriptive, it can guide search towards relatively easy refutations. On the other hand, computing the heuristic involves a certain amount of time and memory overhead. The interleaved structure of the new algorithm may also interfere with the effectiveness of the clause learning and branching heuristic of the underlying solver. The solving performance that is ultimately obtained is due to a combination of these factors.

The procedure is evaluated on the 100 instances used in the Main Track finals of the SAT-Race 2008 competition [1] by comparing the running time of the MiniSat solver to that of a modified solver that uses the aforementioned heuristic. On average and across the entire test set, the modified procedure underperforms the baseline somewhat, but certain classes of satisfiable instances seem to benefit from it.

1.2 Structure

The remainder of this thesis presents some of the problem background, introduces the implicative density as a measure of subproblem solvability, develops an algorithm that uses the density as a heuristic, and describes the measurements used to evaluate its performance. Chapter 2 provides an overview of the Boolean satisfiability problem and of the CDCL solvers that address it, with particular emphasis placed on the details of the most important performance techniques in modern CDCL solvers and on the way that they benefit from the structured nature of industrial SAT instances. Chapter 3 describes implicant cubes, the density heuristic and an algorithm that uses them to guide CDCL search. The procedure is contrasted with the cube-and-conquer algorithm. Chapter 4 describes the measurement setup and results, along with some discussion. The work closes with conclusions in Chapter 5.

[1] A related idea was also proposed by Heule et al. under the name of Cube-and-Conquer [29].

Chapter 2

Foundations of CDCL SAT solvers

Although no single architecture of SAT solver is unsurpassed on every type of problem instance, CDCL solvers could be called the most generally useful solvers today. These solvers perform well on many real-world problems and feature prominently at the top of the rankings in the annual International SAT Solver Competitions [2], in which state-of-the-art SAT solvers race to solve a wide variety of benchmark instances. Innovative performance techniques, such as machine learning-based methods [36, 38], continue to push the boundaries of solver capabilities.

At its core, a CDCL solver systematically searches the domain of a Boolean formula for a satisfying assignment. The procedure is essentially that of the DPLL Algorithm [17], conducting a depth-first search in a binary decision tree, trying out the possible assignments to a specific variable at each node. A naive search down to the leaves would simply enumerate the entire exponentially-sized assignment space, but a great deal of work can be avoided by a host of clever techniques, from the unit propagation procedure originating from the DP Algorithm [16] on which DPLL was based to cutting-edge refinements like those that underpin current competition-winning solvers. The namesake technique of CDCL solvers, conflict-driven clause learning, seeks to learn new, non-obvious implicates of the problem formula by analyzing conflicts during search. Empirical results indicate that CDCL represents a particularly potent contribution to solver performance [33].

The foundation for modern CDCL solvers was laid near the turn of the millennium by the GRASP [40] and Chaff [43] solvers. Many other solvers have since been written around the ideas they introduced. Among the best-known is MiniSat [19], which is used as the baseline solver in this work. MiniSat is designed to be small, extensible and a suitable starting point for research and applications, while also delivering good performance. Although MiniSat itself no longer vies for the top spots in the SAT competitions,

many well-placing solvers are based on GLUCOSE [8], a close descendant.

This chapter reviews the context and foundations of CDCL solvers, as well as of SAT itself. Section 2.1 describes the form, properties, and terminology of SAT, as well as its broad range of applications. Section 2.2 defines the conjunctive normal form (CNF), in which SAT instances are conventionally represented. Section 2.3 defines unit propagation, an efficient procedure for deriving certain consequences from a CNF formula. Section 2.4 introduces the DPLL algorithm, which forms the basis for CDCL solvers. Section 2.5 describes the clause learning technique itself. Section 2.6 discusses the impact of branching heuristics and introduces the variable state independent decaying sum (VSIDS) heuristic. Section 2.7 gives an overview of other, miscellaneous techniques used to boost SAT solver performance. Finally, Section 2.8 reviews research on what characteristics make some SAT instances easy and others difficult. More detailed treatments of most of these topics can be found in [11].

2.1 The Boolean satisfiability problem

Consider a propositional formula F of N variables,

$$F : \{0,1\}^N \to \{0,1\}. \tag{2.1}$$

Each variable can be *assigned* a truth value of either 1 (true) or 0 (false) or may be *unassigned*. The term *assignment* can refer either to the assignment of a single variable or to a (non-contradictory) combination of such assignments. An assignment that specifies a value for every variable is called a *complete assignment* and causes F to evaluate to a truth value. If the value is 1, F is *satisfied*. Otherwise, it is *unsatisfied*. F is called *satisfiable* if there exists at least one assignment that satisfies it; otherwise, F is *unsatisfiable*. The objective in SAT is to determine whether a given formula of this kind is satisfiable or not and, if it is, to produce a satisfying assignment.[1] It is a natural formalism for a wide variety of problems that can be specified by constraints expressed in propositional logic.

One common class of use cases involves proving the tautologyhood of a formula ϕ by demonstrating that its negation $\neg\phi$ is a contradiction (unsatisfiable). This allows SAT to be used for automated theorem proving; indeed,

[1] This is the modern form of the problem, and the one which current solvers address. It is also possible to state SAT as a decision problem, in which case there is no requirement to produce an assignment in the satisfiable case. Knowing such an assignment is often desirable in practice, though, and emerges naturally from the algorithms, so there is no reason to maintain this restriction.

early literature often refers to the problem in these terms [15, 17]. The same principle can be applied to formal verification: given a propositional-logic representation of some system of interest, such as a Boolean circuit, one appends to it a set of additional constraints, such as its specified behavior. The objective is then to demonstrate that the resulting formula is a tautology, which can be shown by proving that its negation is unsatisfiable. On the satisfiable side, one can formalize a system with a number of open choices and the specification for the behavior of that system. A satisfying assignment then describes a set of choices that makes the system behave according to the specification (a determination of unsatisfiability, of course, indicating that no such choice exists).

The Boolean satisfiability problem is NP-complete. In fact, it was the first problem for which this property was proven: Cook [15] showed that, given a problem in NP and a non-deterministic Turing machine that solves it in polynomial time, one can formalize the identification of its accepting path of execution as a SAT instance whose size is polynomial in the length of the path. Since the composition of polynomials is a polynomial, a polynomial algorithm for SAT would give a polynomial algorithm for the original problem, and hence NP-completeness emerges.

SAT is thus placed in a category of problems for which no efficient algorithms are known. Indeed, it is commonly (though not universally) thought that no such algorithm is possible [22]. The expectation of precipitous difficulty, however, only strictly applies in the worst case, and problem instances arising from real-world applications are usually much more tractable than their mere size would suggest, even if they cannot justifiably be called "easy". Such instances typically have an abundance of internal structure, often inherited from the structure of the systems they encode, which a sophisticated solver can exploit to great effect. Consequently, though limited to superpolynomial time in the worst case, SAT solvers are nevertheless effective on many practical problems.

2.2 Conjunctive normal form

SAT instances are usually expressed in conjunctive normal form (CNF) [45]. This is the standard representation in the literature concerning SAT, as well as the one used by most solvers, which typically accept CNF formulas expressed in DIMACS format [32]. The simple linear structure of CNF makes the conditions for satisfiability and unsatisfiability intuitively obvious and helps in devising manipulation rules that can be applied mechanically by computers.

The basic constituent parts of a CNF formula are *literals* and *clauses*. A literal is either a variable or the negation of a variable,

$$l_{ij} \in \{x_1, \ldots, x_N, \overline{x_1}, \ldots, \overline{x_N}\}, \tag{2.2}$$

and a clause is a disjunction of literals,

$$C_i = l_{i1} \vee l_{i2} \vee \ldots \vee l_{iw_i}, \tag{2.3}$$

where N is the number of variables and w_i the number of literals in the ith clause. (Conversely, a *cube* is a conjunction of literals.) The satisfaction of literals, clauses, and cubes is defined in the same way as for propositional formulas: each is satisfied under any assignment that causes it to evaluate to 1. A CNF formula is a conjunction of M clauses,

$$F(x_1, \ldots, x_N) = C_1 \wedge C_2 \wedge \ldots \wedge C_M. \tag{2.4}$$

For instance, the condition that exactly one of 3 variables is true can be written in CNF as

$$(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3}).$$

Alternatively, it is sometimes useful to speak of the conjunction of disjunctions of literals as a set of sets of literals, with equivalent semantics. This makes it possible to describe some useful concepts in terms of set operations: a clause C *contains* a literal l ($l \in C$) if l is one of the terms of the disjunction C and a CNF formula F *contains* a clause C ($C \in F$) if C is one of the terms of the conjunction F.

The interpretation of a CNF formula is straightforward: it is satisfied when all its clauses are satisfied. The empty CNF is a tautology (the conjunctive identity) and the empty clause is a contradiction (the disjunctive identity). From the perspective of constraints on the space of assignments, one may note that any clause is the negation of the cube over the complements of its literals,

$$(l_1 \vee \ldots \vee l_w) \leftrightarrow \neg(\overline{l_1} \wedge \ldots \wedge \overline{l_w}), \tag{2.5}$$

and each clause can be seen as disallowing a particular region of the space of assignments. The totality of what is disallowed is the union of these regions, and the formula is satisfiable if and only if the disallowed region is not equal to the entire assignment space.

2.2.1 The unit clause rule

A CNF formula may contain some number of clauses containing only a single literal, called *unit clauses*, either by construction or as a consequence of preceding manipulation steps. Since all clauses must be satisfied for the formula to be satisfied, each unit clause effectively forces the variable assignment that satisfies its solitary literal, a principle known as the *unit clause rule* [16]. The consequences of unit clauses are usually derived by the efficient unit propagation procedure described in Section 2.3.

2.2.2 The resolution rule

Given two clauses of a CNF formula such that some variable x occurs as the literal x in one and as the literal $\overline{x}$ in the other, a new clause can be derived as their logical consequence [16, 46]:

$$\begin{aligned}(x \vee k_1 \vee \ldots \vee k_n) \wedge (\overline{x} \vee l_1 \vee \ldots \vee l_m) \\ \rightarrow (k_1 \vee \ldots \vee k_n \vee l_1 \vee \ldots \vee l_m).\end{aligned} \tag{2.6}$$

This is known as *resolution* on x, and the implicate is known as the *resolvent* of the two clauses. The resolvent is non-trivial if and only if resolution between the clauses was only possible on a single variable; otherwise, it contains some other variable in both polarities and is, therefore, a tautology. The resolvent itself can participate in further resolutions, making long chains of deduction possible.

In the event that resolution is possible on two unit clauses, the result is an empty clause, equivalent to a contradiction:

$$(x) \wedge (\overline{x}) \quad \rightarrow \quad \bot. \tag{2.7}$$

Any sequence of resolutions that ends this way, called a *resolution refutation*, ultimately proves that the original formula was unsatisfiable. In fact, for any unsatisfiable formula, a refutation by resolution exists [46], though the minimum size of that refutation (the number of clauses it contains) is exponential in the problem size in the worst case [49].

2.2.3 Encoding formulas to CNF

An arbitrary propositional formula F of N variables can be converted to a CNF formula F_{CNF}, possibly introducing some number N' of new variables, such that F_{CNF} preserves the satisfiability of the original formula (as far as the original set of variables is concerned) [45]:

$$F(x_1, \ldots, x_N) = F_{\text{CNF}}(x_1, \ldots, x_N, z_1, \ldots, z_{N'}). \tag{2.8}$$

$$g(x_1, x_2, x_3) = \underbrace{(x_1 \wedge x_2)}_{z_1} \rightarrow x_3$$

$$\begin{aligned} z_1 \leftrightarrow (x_1 \wedge x_2) \quad &\longrightarrow \quad (\overline{z_1} \vee x_1) \wedge (\overline{z_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee z_1) \\ z_1 \rightarrow x_3 \quad &\longrightarrow \quad (\overline{z_1} \vee x_3) \end{aligned}$$

$$g_{\mathrm{CNF}}(x_1, x_2, x_3, z_1) = (\overline{z_1} \vee x_1) \wedge (\overline{z_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee z_1) \wedge (\overline{z_1} \vee x_3)$$

Figure 2.1: A Tseitin encoding of a simple formula.

The conversion can, in principle, be achieved by naive application of Boolean algebra, requiring the introduction of no new variables. However, this approach can produce unmanageably large formulas. Typically, one instead uses an encoding like the one described by Tseitin [49], an example of which is shown in Figure 2.1. This encoding associates each subformula of F with a new variable and expresses the equivalence between the two in clausal form. It is relatively compact; if F was expressed in terms of $\neg$, $\vee$, $\wedge$, $\rightarrow$, and $\leftrightarrow$, the size of F_{CNF} is linear in the size of F.

Also useful are certain encodings designed to express specific kinds of constraints. For instance, the constraint that at most one out of n literals may be true can be conveniently expressed with the so-called ladder encoding [23], which requires $O(n)$ clauses and introduces $O(n)$ new variables. Expressing the constraint by pairwise enumeration, by contrast, requires $O(n^2)$ clauses.

2.3 Unit propagation

As mentioned in Section 2.2.1, any unit clauses that a CNF formula contains force corresponding variable assignments. Once such a variable assignment is known, it is possible to eliminate the associated variable from the formula without affecting satisfiability by applying a few simple rules to each clause in the formula. This procedure, repeated until no unit clauses remain, is called unit propagation (UP) or Boolean constraint propagation (BCP). It originates with the Davis-Putnam (DP) procedure, an early algorithm for theorem proving [16].

If a CNF formula F contains the unit clause (y), propagating the literal y consists of applying the following rules to all clauses (in each case, Λ stands for the clauses not written out explicitly):

1. If a clause contains the literal y, the forced assignment satisfies it, so it is removed from the formula:

$$y \wedge (y \vee \ldots) \wedge \Lambda \quad \leftrightarrow \quad y \wedge \Lambda. \tag{2.9}$$

2. If a non-unit clause contains the literal $\overline{y}$, the forced assignment makes that literal impossible to satisfy, so the clause is replaced by itself with $\overline{y}$ removed:

$$y \wedge (\overline{y} \vee l_1 \vee \ldots) \wedge \Lambda \quad \leftrightarrow \quad y \wedge (l_1 \vee \ldots) \wedge \Lambda. \tag{2.10}$$

3. If a clause is the unit clause $(\overline{y})$, a contradiction is derived:

$$y \wedge (\overline{y}) \wedge \Lambda \quad \leftrightarrow \quad \bot. \tag{2.11}$$

The remaining formula, marked $F|_y$, contains no occurrences of either literal y or literal $\overline{y}$; even the original unit clause (y) is removed by rule 1. The procedure is repeated until all unit clauses have been exhausted, including any new unit clauses derived by previous applications of rule 2. Ultimately, one of the following fixed points is reached:

1. A contradiction has been derived; the formula is unsatisfiable.

2. All clauses have been eliminated; the formula is satisfiable.

   The satisfiable assignment is the combination of the variable assignments forced by unit clauses over the course of unit propagation, with any variable without a forced assignment free to take either value.

3. A non-empty formula with no unit clauses is reached, providing no immediate conclusions about satisfiability.

Unit propagation is an efficient way of deriving certain straightforward consequences of a CNF formula. However, most interesting instances involve more complex interdependencies than it has the capacity to handle, such as in the formula

$$(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}), \tag{2.12}$$

which encodes an exactly-one-of-two condition. Neither x_1 nor x_2 is locked into a specific truth value in all satisfying assignments, and consequently, unit propagation fundamentally cannot eliminate them. Although unit propagation is therefore not complete by itself, it is very useful as a component of a larger solver.

2.3.1 Pure literal elimination

Besides the rules of unit propagation, [16] defines a rule now known as *pure literal elimination*: any variable that occurs in only one polarity is assigned the associated truth value with no loss of satisfiability. The corresponding literal is called *pure*. For a given variable x,

1. If the formula contains the literal x but not the literal $\overline{x}$, then x is a pure literal and assigning $x = 1$ preserves satisfiability.

2. If the formula contains the literal $\overline{x}$ but not the literal x, then $\overline{x}$ is a pure literal and assigning $x = 0$ preserves satisfiability.

In either case, the pure literal is propagated throughout the formula, eliminating all clauses in which it occurs by rule 1. Since rules 2 and 3 are not triggered on a pure literal, this procedure cannot cause a previously satisfiable formula to become unsatisfiable (it may, however, shrink the set of satisfying assignments).

Note that pure literal elimination will never produce new unit clauses, meaning that applying pure literal elimination to a formula closed under unit propagation yields a formula that is also closed under unit propagation. In an algorithm with a simplification step using both unit propagation and pure literal elimination, therefore, pure literal elimination can be done strictly after unit propagation.

2.3.2 The Davis-Putnam procedure

The complete DP Algorithm [16] proceeds from a formula closed under unit propagation and pure literal elimination by a rule for eliminating a variable that is neither pure nor occurs in a unit clause. To eliminate the variable x, the literals x and $\overline{x}$ are first factored out of the clauses in which they appear:

$$(x \vee A_1) \wedge (x \vee A_2) \wedge \ldots \quad \leftrightarrow \quad x \vee (A_1 \wedge A_2 \wedge \ldots) \quad \equiv \quad x \vee A; \tag{2.13}$$

$$(\overline{x} \vee B_1) \wedge (\overline{x} \vee B_2) \wedge \ldots \quad \leftrightarrow \quad \overline{x} \vee (B_1 \wedge B_2 \wedge \ldots) \quad \equiv \quad \overline{x} \vee B. \tag{2.14}$$

Here A and B are CNFs. Marking by R the clauses in which neither x nor $\overline{x}$ appears, the formula is then simplified as follows:

$$(x \vee A) \wedge (\overline{x} \vee B) \wedge R \quad \rightarrow \quad (A \vee B) \wedge R. \tag{2.15}$$

The original formula is satisfiable if and only if the simplified formula is satisfiable. This is equivalent to replacing the clauses containing x and $\overline{x}$ with all possible resolvents on x.

The term $(A \vee B)$ must be expanded using the distributive property in order to again obtain a formula in CNF. In general, this replaces $|A| + |B|$ clauses with $|A||B|$ clauses. Although many clauses in the resulting formula may be redundant (duplicates of earlier clauses) or trivial (containing both polarities of some variable and therefore identically true), this rule can grow the size of the formula quite rapidly in the general case.

2.4 The DPLL algorithm

As it quickly turned out that the DP procedure could cause the formula to become prohibitively large, an alternative rule for proceeding from a formula closed under unit propagation and pure literal elimination was introduced, leading to the Davis-Putnam-Logemann-Loveland (DPLL) Algorithm [17]. Instead of an elimination rule that retains all past its work explicitly in the formula, proceeding linearly, DPLL uses a splitting rule that effectively leads to a treelike search in the space of assignments. Past work is either implicit in the location in the search tree or simply discarded, and the space requirements remain manageable, especially with clever implementation. This basic idea has stood the test of time: the organization of the search in modern CDCL solvers remains essentially that of DPLL.

The *splitting rule* of DPLL, applied to a formula closed under unit propagation and pure literal elimination, is as follows. Consider an arbitrary CNF formula F and a variable x that occurs in F. The formula F can be expressed in terms of the Shannon expansion,

$$F \leftrightarrow (F|_x \wedge x) \vee (F|_{\overline{x}} \wedge \overline{x}), \tag{2.16}$$

and the satisfiability of F can be determined as follows:

1. Determine the satisfiability of $F|_x$.
2. If $F|_x$ is satisfiable, $F|_x \wedge x$ is satisfiable and therefore F is satisfiable.
3. Otherwise, determine the satisfiability of $F|_{\overline{x}}$.
4. If $F|_{\overline{x}}$ is satisfiable, $F|_{\overline{x}} \wedge \overline{x}$ is satisfiable and therefore F is satisfiable.
5. Otherwise, F is unsatisfiable.

The subproblems of finding the satisfiability of the conditioned formulas $F|_x$ and $F|_{\overline{x}}$ are solved one at a time, limiting space requirements. The corresponding literal over x is added to F as a unit clause, unit propagation and

pure literal elimination are applied, and if the result remains indeterminate, the splitting rule is applied again. This continues for as long as the result remains indeterminate:

$$F \leftrightarrow (F|_x \wedge x) \vee (F|_{\overline{x}} \wedge \overline{x}) \tag{2.17}$$
$$\leftrightarrow ([(F|_{x,y} \wedge y) \vee (F|_{x,\overline{y}} \wedge \overline{y})] \wedge x) \vee (F|_{\overline{x}} \wedge \overline{x}) \tag{2.18}$$
$$\leftrightarrow \ldots \tag{2.19}$$

An application of the splitting rule is called a *decision* and the recursion depth is called the *decision level*. The chosen variable is called the *decision variable*.

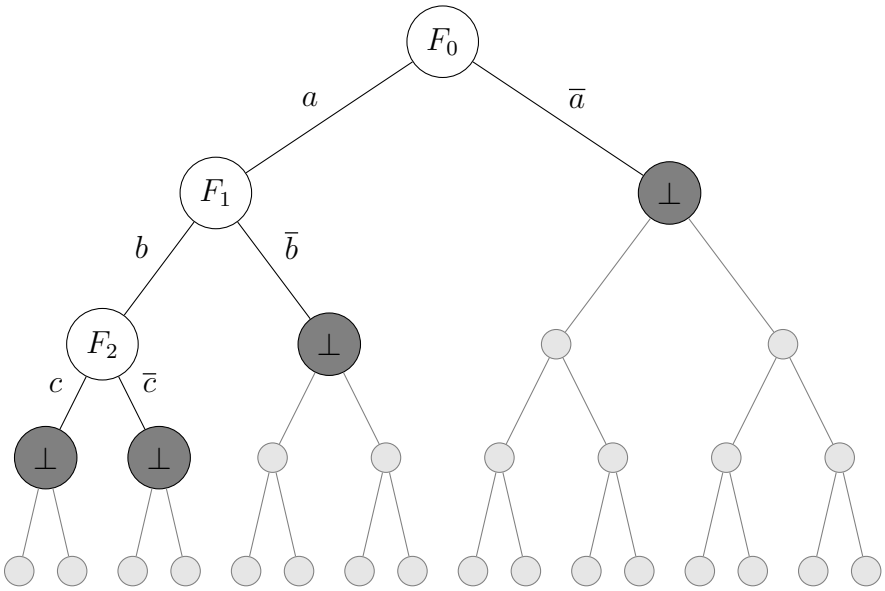
This algorithm can be thought of as a depth-first traversal of a binary tree (Figure 2.2). The outgoing pair of edges of each internal node corresponds to a single application of the splitting rule, and each node is associated with a formula equivalent to F conditioned on the literals specified by its path from the root. The search starts from the root and proceeds until it reaches a node where the satisfiability or unsatisfiability of the associated formula can be determined by unit propagation. If the formula is satisfiable, the search terminates, declaring F to be satisfiable. Otherwise, it *backtracks* to the last node that still has unexplored children. Once each child of the root has been fully explored without finding a satisfying assignment, F can be declared unsatisfiable.

The depth of the tree is the number of variables N in the CNF formula, and its 2^N leaves correspond to all possible assignments. The formula associated with a leaf node is just its value under that assignment, so its satisfiability is determined by simple evaluation. As the choice of the splitting variable is not specified, the tree is not unique. Likewise, sibling branches are not required to proceed in the same order.

The running time of DPLL is dominated by the number of applications of the splitting rule. Whenever the formula at a node can be refuted by unit propagation, all of its descendant splits can be avoided, as can the splits corresponding to those variables eliminated by unit propagation on the path from the root.[2] The depth of nodes refutable by unit propagation is highly dependent on the choice of branching variable, and a good variable choice heuristic significantly improves the performance of DPLL. These heuristics are briefly discussed in Section 2.6.

[2]When a node is instead found satisfiable by unit propagation the algorithm can terminate immediately, so the benefit in question is primarily relevant in refutations.

$$F_0 = (a \vee b) \wedge (a \vee \overline{b}) \wedge (\overline{a} \vee c \vee \overline{d}) \wedge (b \vee \overline{c}) \wedge (\overline{b} \vee \overline{c} \vee d) \wedge (c \vee d) \wedge (\overline{c} \vee \overline{d})$$



$$F_1 = F_0|_a, \qquad F_2 = F_1|_b.$$

Figure 2.2: An example of a DPLL tree of a simple unsatisfiable formula, with splitting variables chosen in numerical order. Dark gray nodes are where unit propagation first derives a contradiction. Light gray nodes inherit $\bot$ from their parents.

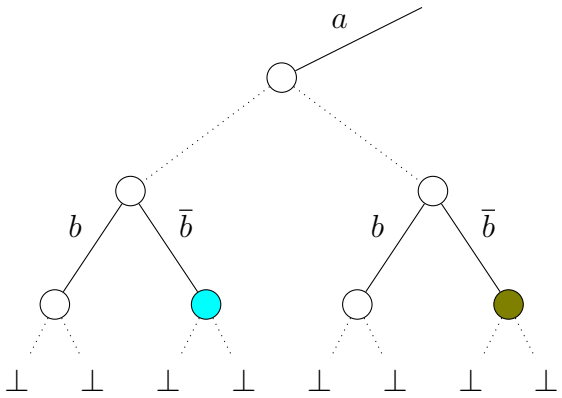


Figure 2.3: An illustration of the problem of redundant refutations. Consider the DPLL tree of a formula that conflicts with the cube $(a \wedge \overline{b})$ in a way that is not obvious from the original clauses. This cube is necessarily contained in the partial assignments associated with both the cyan and the olive green node, with the consequence that all paths descending from these nodes must ultimately end in conflicts, quite possibly for at least partly the same reasons. If the search procedure could summarize and remember the reasons for the conflict at the cyan node, it might be able to avoid repeating some or all of the effort at the olive green node.

2.5 Conflict analysis and clause learning

As shown in equation (2.5), each clause in a CNF formula specifies a cube of variable assignments under which the formula is necessarily unsatisfiable. However, not every cube that conflicts with the formula is apparent from one of its clauses in this way. When the DPLL algorithm ends up on a path with a partial assignment containing such a non-obvious conflicting cube, it may need to branch several times before a refutation is obtained. Since the conflicting cube can occur on multiple paths, the work of refuting it is often repeated many times, as illustrated in Figure 2.3. This kind of redundant effort is a significant drag on performance, motivating the use of conflict analysis to learn new clauses that summarize the insight from spent work. A technique for achieving this was introduced by the solver GRASP [40]. Now known as conflict-driven clause learning [41], it gives the name to modern CDCL solvers.

At each node of the DPLL decision tree, the search can determine a number of mandatory variable assignments, either directly chosen by an upstream branching decision (*decision assignments*) or derived during unit propaga-

$$F = \ldots \wedge \underbrace{(\overline{x_3} \vee x_4)}_{\omega_1} \wedge \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_4})}_{\omega_2} \wedge \underbrace{(\overline{x_1} \vee \overline{x_5} \vee x_6)}_{\omega_3} \wedge \underbrace{(x_2 \vee \overline{x_5})}_{\omega_4} \wedge \ldots$$

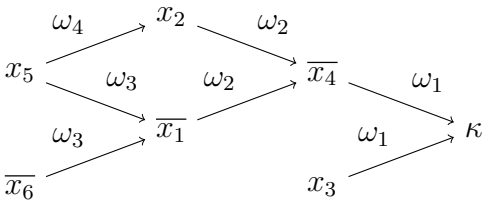


Figure 2.4: The implication graph corresponding to a conflict in a branch with a partial assignment containing the cube $(x_3 \wedge x_5 \wedge \overline{x_6})$.

tion (*implied assignments*). In the latter case, the clause that became unit to imply a given variable is the *antecedent clause* of that variable, and the *antecedent assignments* of the implied assignments are the ones that collectively forced the antecedent clause to become unit:

$$(l_1 \vee \ldots \vee l_n) \quad \leftrightarrow \quad ((l_1 \vee \ldots \vee l_{n-1}) \vee l_n) \tag{2.20}$$
$$\leftrightarrow \quad \neg(l_1 \vee \ldots \vee l_{n-1}) \rightarrow l_n \tag{2.21}$$
$$\leftrightarrow \quad (\overline{l_1} \wedge \ldots \wedge \overline{l_{n-1}}) \rightarrow l_n. \tag{2.22}$$

CDCL also keeps track of the *decision level* of each assignment, meaning the depth of the search tree at the node where the assignment was decided or derived. The sufficient conditions for each assignment are recorded in an *implication graph* (Figure 2.4). Each assignment is a vertex in the graph, and there is a directed edge to any implied assignment from its antecedent assignments, labeled by the antecedent clause.

If unit propagation derives a conflict, it is associated with a special vertex κ of the implication graph, and its antecedent assignments are the assignments that force the antecedent clause to become conflicted rather than just unit:

$$(l_1 \vee \ldots \vee l_n) \quad \leftrightarrow \quad \neg(l_1 \wedge \ldots \wedge l_{n-1}) \rightarrow \bot. \tag{2.23}$$

Given an implication graph containing a conflict, one can derive a number of partial assignments (cubes) that are sufficient to produce that conflict, the negation of any such cube being a clause that is an implicate of the formula. The procedure starts with the negation of the antecedent clause of κ. From there, identifying each assignment with the corresponding literal, each literal in the cube can be replaced with its antecedent literals, until all the literals correspond to decision assignments. This is equivalent to a

repeated application of resolution to the sequence of the antecedent clauses of the assignments until all variables in the clause are decision variables. At that point, there are no more antecedents with which to resolve, and the algorithm *learns* the resulting clause (appends it to the formula). An example analysis is shown in Figure 2.5.

2.5.1 Unique implication points

This analysis proceeds up to the decision variables, but in fact, the clause under development is a valid implicate of the formula after each resolution step, not just at the end. One can, therefore, choose to learn a clause corresponding to an earlier cut of the implication graph. A common choice is to not continue analysis past any unique implication points (UIP) [40], implied-variable vertices that *dominate* some number of decision-variable vertices (i.e., every path from a dominated vertex to κ must go through the UIP). The aim of this is to preferentially learn shorter clauses, which represent stronger information about the problem.

2.5.2 Backjumping

Once a conflicting cube has been identified, it is apparent that no satisfying assignment is possible until at least one of the cube literals is undone by backtracking. In practice, since the active branch cannot be reordered, one must consider the assignment that was most recent in the decision tree. The search is free to backtrack to the decision level of that assignment (the *backtrack level*) immediately. If the distance to the backtrack level is more than a single level, this is called *non-chronological backtracking* or *backjumping*. In the case of a conflict that immediately follows from an unforced assignment, the backtrack level is necessarily the immediately preceding level, but on successive conflicts, a backjump may be possible.

2.5.3 Managing the clause database

As search progresses from conflict to conflict, the database of conflict clauses keeps growing. If left unchecked, this may eventually exhaust all of the available memory. Additionally, although the addition of even quite long clauses (dozens of variables) can improve solver performance, the degree to which learning them represents a tradeoff increases with clause size [40]. These considerations indicate that a clause database management policy is a useful feature for a sophisticated solver. As simple options, one may limit the size of learnt clauses, as in GRASP [40], or delete clauses once a given

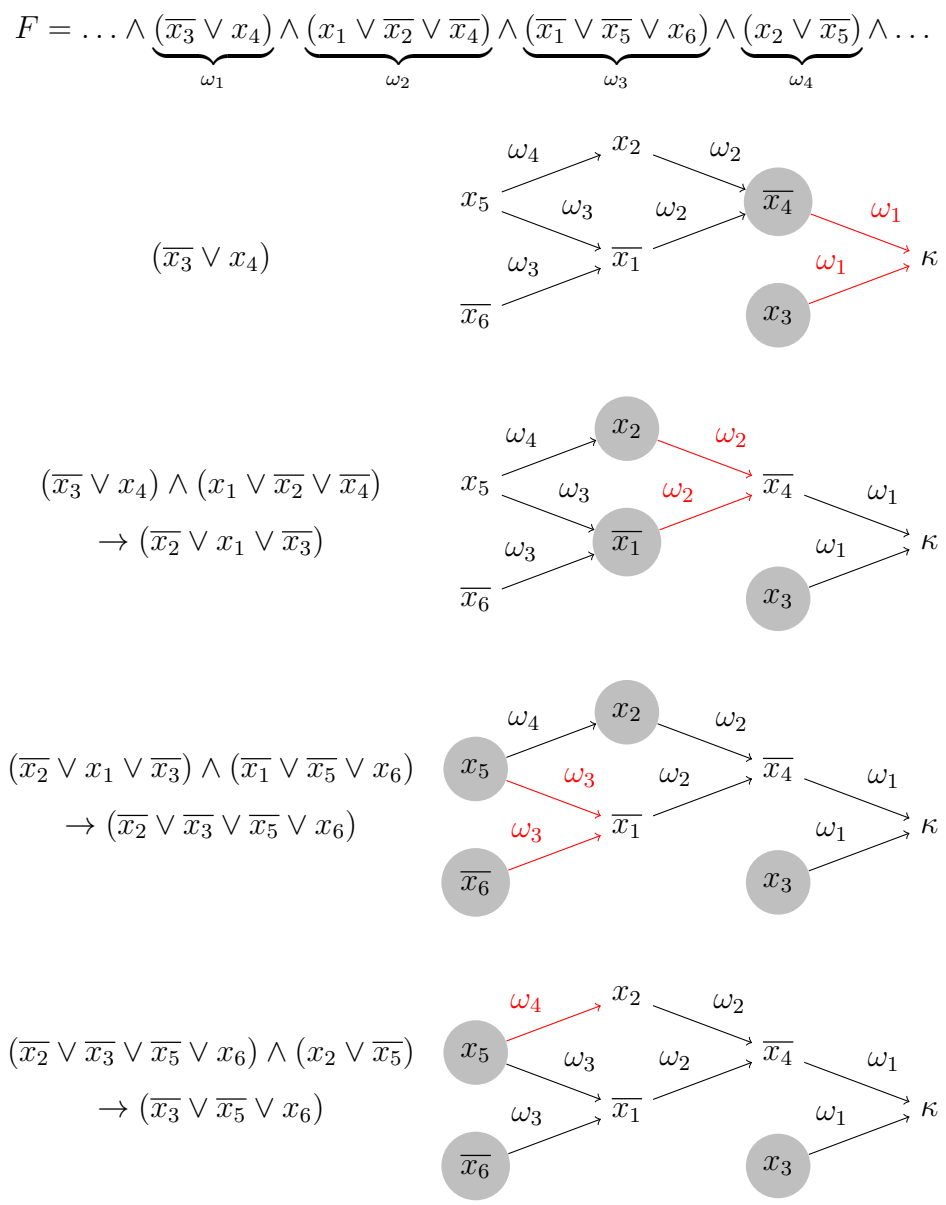


Figure 2.5: A possible derivation of a conflict-induced clause in the implication graph of Figure 2.4, with the corresponding sequence of resolutions. At each step, the conjunction of the literals marked with a gray circle is a sufficient condition for a conflict, and its negation is a valid implicate of F.

number of their literals become unassigned, as in CHAFF [43] (note that, at the moment of creation, *all* the literals of a learned clause are unsatisfied). Alternatively, a heuristic approach may be used, as in MINISAT, which uses an activity heuristic for the learnt clauses [48].

2.6 VSIDS and other branching heuristics

The DPLL algorithm specifies the general structure and traversal paradigm of the search tree, but not any particular method for choosing the branching variable at any given decision level. The branching choice has a dramatic influence on the work required to solve the problem instance. This is immediately evident in the case of satisfiable instances, which can ignore vast regions of the search space if steered early into a satisfying assignment by an insightful decision method, but both satisfiable and unsatisfiable instances benefit from methods that are good at finding decisions with shallow refutations, as this reduces the effective depth of the search. In practice, algorithms choose branching variables based on *branching heuristics*, many of which can be viewed from the perspective of estimating the refutation work. However, the choice of branching heuristic involves balancing the benefit of branching into simpler subproblems with the cost of computing a more complicated heuristic, and the heuristic that gives the best splits is not necessarily the same as the one that gives the best performance.

The most prominent heuristic in current CDCL solvers is the variable state independent decaying sum (VSIDS) [43]. In its original form, VSIDS maintains a value called *activity* for each polarity of each problem variable (for every variable x_i, one activity value for x_i and one for $\overline{x_i}$). The activity of a given literal measures its tendency to occur in a conflict clause. On learning a clause, the solver increments (*bumps*) the activity of each literal in the clause by a constant value. The activities are periodically multiplied by a *decay factor*, typically after each conflict. The next branching assignment is chosen to match the literal with the highest activity. MINISAT uses a variant of VSIDS that does not make the distinction between polarities, maintaining a single activity value per problem variable [19].

Intuitively, VSIDS is compatible with the notion that a variable that is relevant (acts as a potential blocker) in one branch is likely to also be relevant in other branches, especially nearby branches (more recent, less decayed ones). This idea is supported by experimental results [37] indicating that VSIDS tends to choose structurally important variables (bridge variables in the variable incidence graph; this is discussed in more detail in Section 2.8). VSIDS is also distinguished by its low overhead. It remains a cornerstone

heuristic today, though alternative heuristics continue to be proposed and implemented [10, 24, 26, 47]; for instance, the CaDiCaL solver [9] that has done well in recent SAT competitions uses a heuristic called variable move-to-front (VMTF) [47].

2.7 Other performance techniques

The impressive performance of modern CDCL solvers is the result of a combination of techniques. One rather comprehensive empirical study [33] evaluated the importance of four prominent ones, measuring the impact on running time caused by disabling each individual feature in a modified version of MiniSat. In decreasing order of performance impact, these techniques were clause learning (Section 2.5), the VSIDIS heuristic (Section 2.6), two-watched-literals and search restarts. This section briefly discusses the latter two.

2.7.1 Optimized unit propagation using two-watched-literals

Most of the running time of a CDCL solver (on the order of 90%) is spent on unit propagation, making it a rewarding target for optimization. The trivial approach of simply iterating through the clauses clearly wastes an enormous portion of its time looking at clauses that are not affected by the constraint that is being propagated. The approach that immediately suggests itself is to maintain a lookup table from each literal y to a list of clauses containing its negation $\overline{y}$ (clauses that are shrunk by the assignment corresponding to y). In fact, it is possible to do even better. Note that the shrinking of a clause is of practical interest only when it becomes unit, producing an implied assignment. The authors of Chaff [43] noted that to detect this condition, it is enough to monitor only two of the non-zero literals in each clause at a time (these are called *watched literals*). Any assignment that reduces a clause to unit size must necessarily set one of the watched literals to 0, therefore it is only necessary to check the clause size when that happens. If the clause did not become unit, we find a non-watched literal that is not set to 0 and replace the watched literal that became 0 with it, maintaining the condition of two non-zero watched literals. The two-watched-literals technique is visualized in Figure 2.6.

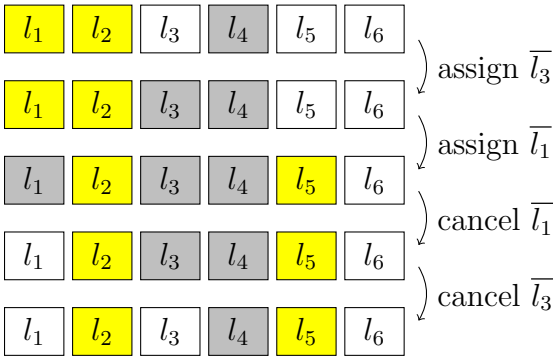


Figure 2.6: Unit propagation using the two-watched-literals optimization technique, from the perspective of a clause $(l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6)$. Literals that evaluate to zero under the assignment are indicated with a gray background and watched literals with a yellow one.

2.7.2 Restarts

The running times of combinatorial search methods, such as the DPLL-like search that underpins CDCL solvers, are known for high variability (their running times can have so-called *heavy-tailed distributions*) [27]. Even a small change in initial conditions, up to the labeling of the variables or the choice of a random seed, can have a dramatic effect on performance. Through randomization methods, an algorithm can effectively "hedge" across multiple runs, significantly improving outcomes [28]. CDCL solvers take advantage of this technique by restarting at the root of the search tree once a certain number of conflicts have been encountered (while retaining the database of learnt clauses, as well as other accumulated information such as variable activities). Intuitively, restarts give the search an opportunity to escape from a pathologically difficult region of the search space, or just to order its branching decisions in a way that promotes shallower refutations.

The way the cutoff for the number of conflicts changes with the number of restarts is specified by the *restart policy*, here expressed as a sequence $\mathcal{R} = (t_1, t_2, \ldots)$ of *restart intervals* (t_i is the number of conflicts that the ith run is permitted to encounter before a restart occurs). Some options for a restart policy are:

1. A fixed-interval policy: the restart interval is a constant value C_0,

$$\mathcal{R} = (C_0, C_0, C_0, C_0, \ldots). \tag{2.24}$$

2. A geometric policy: the restart interval grows by a constant factor γ

after each restart,

$$\mathcal{R} = (C_0, \gamma C_0, \gamma^2 C_0, \gamma^3 C_0, \ldots). \tag{2.25}$$

3. A Luby sequence-based policy: the restart interval evolves according to the Luby sequence,

$$t_i = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1, \\ t_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1. \end{cases} \tag{2.26}$$

This is proven to be an optimal *universal strategy* for restarts of a *Las Vegas algorithm* (an algorithm whose running time is a random variable), that is, it is optimal when the distribution of the running times is not known [39]. The start of the sequence is:

$$\mathcal{S} = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 4, 8, 1, \ldots). \tag{2.27}$$

Rather than using the sequence directly, it is more practical to use it to multiply some base interval C_0:

$$\mathcal{R} = (C_0, C_0, 2C_0, C_0, C_0, 2C_0, 4C_0, \ldots). \tag{2.28}$$

4. An exponential Luby sequence policy: the restart interval is proportional to an exponential function with a fixed base k and an exponent given by the Luby sequence,

$$\mathcal{R} = (kC_0, kC_0, k^2C_0, kC_0, kC_0, k^2C_0, k^4C_0, \ldots). \tag{2.29}$$

MiniSat 2.1 uses this kind of policy by default, with an option to use a geometric policy instead.

Several different restart policies (three fixed-interval policies, two geometric policies, and a Luby policy with a base interval of 32) were empirically compared to each other and to a solver without restarts in [30]. All policies improved on the no-restarts baseline, with the Luby policy performing particularly well.

2.8 Relative difficulty of SAT instances

When evaluating and improving on methods for solving SAT, it would be useful to know what makes some problem instances easy and others hard. Intuitively, we can expect problems with extensive internal structure to be

more tractable than those with little or no internal structure. Making general predictions about instance hardness is far from trivial, but there exists a body of research using, among others, statistical and network analysis methods to shed some light on the relative difficulty of different SAT instances. This section provides a brief outline of some of this research.

2.8.1 Random k-CNF formulas

Statistical and algebraic analysis of the generalities of SAT benefits greatly from a rigorously defined family of instances that is amenable to liberal sampling. Historically, random k-CNF formulas have been widely used for this purpose. These instances lack structural features that are decisive to the tractability of industrial problems (see Section 2.8.2) and can therefore hardly be called "typical". Nevertheless, they expose some interesting insights, particularly on the asymptotic properties of SAT instances.

Given a choice of n variables, there are a total of $2^k\binom{n}{k}$ unique, non-trivial clauses of k literals: $\binom{n}{k}$ combinations of k literals and 2^k choices of polarities [3]. The subsets of m such clauses form a family of instances, parametrized by (k, n, m). The *density* of a family, $r = m/n$, turns out to act as a kind of intensive property, dividing the set of k-CNF instances of n variables into two regimes: for small r, instances are likely to be satisfiable; for large r, they are likely to be unsatisfiable. In experiments, there appears to be a clear threshold around which the transition from the satisfiable to the unsatisfiable regime occurs. For $k = 3$, the point of equal probability occurs around $r \approx 4.2$ [34, 42]. It is conjectured that, in the limiting case of $n \to \infty$, the threshold occurs precisely at some critical density depending only on k, analogous to a phase transition. The existence of the critical value has not been proven, though there do exist upper and lower bounds for r, beyond which the probability that the satisfiability is the one typical of the regime approaches certainty [14, 21, 35].

The empirical difficulty of 3-CNF formulas, as measured by the number of recursive calls of a DPLL algorithm, has been found to spike as the density of the formula approaches the threshold where 50% of instances are satisfiable, then fall off with further increases [42]. The experimenters related the relative easiness at low and high densities to the formulas being *underconstrained* or *overconstrained*, respectively. (Interestingly, sparse formulas were only easy on average, as they were predominantly satisfiable. Sparse formulas that were nevertheless unsatisfiable were actually *more* difficult than dense ones, just quite rare.) Theoretical results for general k, in the limit $n \to \infty$, support this picture. For instance, it is (asymptotically) easy to find a satisfying assignment for a random k-CNF formula given r below a certain threshold

value ($r < \frac{2}{3}$ in the case of $k = 3$); specifically, a linear algorithm finds such an assignment with probability tending to 1 as $n \to \infty$ [12]. Conversely, for $r \geq 0.7 \times 2^k$, the shortest refutation by resolution involves an exponential number of clauses in the asymptotic case, implying that solving it by DPLL requires an exponential number of steps [13].

2.8.2 Topological properties of industrial instances

An intuitive way for examining the structure of a SAT instance is in terms of the topology of a graph representation. Two common graph representations are:

1. The bipartite graph, in which there is a node for each variable and each clause and an edge for each occurrence of a variable in a clause; and

2. The variable incidence graph, in which there is a node for each variable and an edge for each pair of variables that occur together in a clause.

These graph representations are illustrated in Figure 2.7. Broadly, the closeness of two vertices in each kind of graph gives some indication of how closely the fates of the variables or clauses they represent are linked.

Contrary to random formulas, industrial SAT instances often have structural properties similar to the ones that occur in networks studied by other domains. Many benchmarks, for instance, exhibit a power-law distribution in vertex arity [4, 7], similar to what exists for example in the topology of the Internet [20], indicating a *scale-free structure*. Industrial instances also have high *modularity* [6], that is, they are structured into communities (broadly, groups of vertices that tend to connect to each other more than to outside vertices, by analogy to human communities), and the quality of the community structure (its Q value) actually correlates with solver performance [44]. Not surprisingly, the VSIDS heuristic has been found to favor topologically important variables [37]. Besides helping us understand the performance of existing solver techniques and design new ones, understanding the topological properties of industrial instances gives us information based on which to generate more realistic random SAT formulas [5, 25].

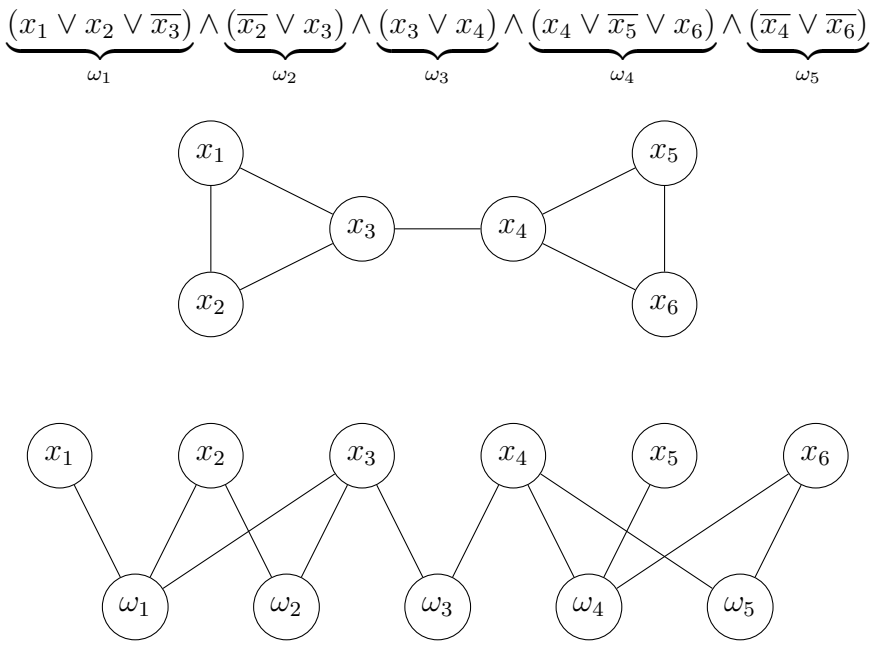


Figure 2.7: Variable incidence graph (top) and bipartite graph (bottom) representations of a simple CNF formula.

Chapter 3

Contributions

The ordinary DPLL-like solving procedure for a CNF formula is analogous to a depth-first search in a decision tree. The precise choice of the search path is normally made with the help of a heuristic such as VSIDS, and the performance improvements obtained through a good heuristic illustrate the pivotal importance of good branching decisions. One way to further improve the choice of search path might be to apply a degree of breadth-first reasoning to the top-level choices made by the algorithm. The central contribution of this thesis is an application of this principle.

This kind of avenue of attack for SAT has previously been investigated by Heule et al., [29] in the form of the Cube-and-Conquer solving method. Cube-and-Conquer solves a SAT instance in two stages:

1. First, a lookahead solver is used to make a shallow breadth-first traversal of the decision tree, dividing the space of variable assignments into regions with heuristically favorable difficulty characteristics. This amounts to a partition of the space. Each resulting subproblem is identified with the cube of decision assignments by which it is reached from the problem root.

2. Subsequently, the subproblems are solved by CDCL solvers, in an application of the divide-and-conquer paradigm.

The logic is to apply a lookahead solver at the top of the tree, where the problem is expected to be "hard" and to especially benefit from sophisticated branching decisions, but a CDCL solver at the lower levels, where the problem is expected to be "easy" and to especially benefit from rapid search. More generally, Cube-and-Conquer is a case of dividing the search space, which opens possibilities for parallel or distributed solving; this is discussed in detail for example in [31].

This thesis presents one approach to using problem subcubes to guide search. Where Cube-and-Conquer is exhaustive, the present method is opportunistic: rather than cubing down to a strict partition which has favorable characteristics as a whole and synthesizing the result from the results of the subproblems, it looks for individual cubes with individually favorable characteristics and attempts to derive strong clausal implicates from their refutations, ensuring completeness by interleaving with a search of the unconditioned problem.

This chapter formally presents the method that is the topic of the thesis, addressing the scoring method used to determine which cubes it attempts to refute, the choice of the cubes for which scores are computed in the first place, and the composition of these into an algorithm, based on a modification of MiniSat 2.1.

3.1 Criteria and methods for cube selection

3.1.1 Implicative density

The cube-scoring method used here is consistent with using the number of unbound variables as a proxy for the difficulty of a particular subproblem. This is intuitively reasonable, since one expects this number to correlate with the depth of the search tree, and therefore also with the refutation work. In this view, given some CNF formula closed under unit propagation, a particular cube A can be considered easier than the unconditioned problem by the total number of variable assignments that it generates under unit propagation, including both decision assignments and implied assignments. We say here that A *yields* such assignments, and denote their number as $y(A)$; for cubes refuted by bare unit propagation, we say that A *yields a conflict*, and leave $y(A)$ as undefined. Larger cubes naturally tend to yield more numerous assignments, yet their refutations also derive weaker constraints, meaning that $y(A)$ by itself is not a practical scoring method. Instead, we correct for the cube size by measuring yield per decision, here called (implicative) *density*,

$$\rho(A) = \frac{y(A)}{|A|}. \tag{3.1}$$

3.1.2 Implicant cubes

In principle, one can compute the density for any cube that does not yield a conflict. Out of the cubes small enough to be of practical interest, however, typically only a fraction actually yields new implied assignments (and is

therefore indicated as possibly advantageous to search). Moreover, the full set of all possible cubes, even up to quite a limited size, very quickly becomes infeasible to enumerate. Fortunately, for a given CNF formula closed under unit propagation, we have a reasonable option of restricting ourselves to a particular, comparatively small set of cubes $\mathcal{I}$, here called the *implicant cubes* or *implicants*:

$$\mathcal{I} = \{\neg(X|_{\overline{\ell}}) \mid X \text{ is a non-unit problem clause and } \ell \in X\}. \tag{3.2}$$

Let us refer to the relationship between a clause X, one of its literals ℓ, and the implicant $\neg(X|_{\overline{\ell}})$ as *association*.

Any implicant $\neg(X|_{\overline{\ell}})$ that does not cause unit propagation to derive a conflict necessarily causes it to derive the assignment ℓ, since propagating the implicant reduces the associated clause to a unit clause:

$$(l_1 \vee l_2 \vee l_3 \vee \ldots \vee \ell)|_{\overline{l_1}} = (l_2 \vee l_3 \vee \ldots \vee \ell) \tag{3.3}$$
$$(l_1 \vee l_2 \vee l_3 \vee \ldots \vee \ell)|_{\overline{l_1},\overline{l_2}} = (l_3 \vee \ldots \vee \ell) \tag{3.4}$$
$$\vdots$$
$$(l_1 \vee l_2 \vee l_3 \vee \ldots \vee \ell)|_{\overline{l_1},\overline{l_2},\overline{l_3},\ldots} = (\ell). \tag{3.5}$$

Unit propagation therefore necessarily finds at least one implied assignment (namely, ℓ). If unit propagation does derive a conflict, the implicant is inconsistent with the formula, and its negation $X|_{\overline{\ell}}$ is a valid clausal implicate of the formula. Because $X|_{\overline{\ell}} \rightarrow X$, the problem clause X can be strengthened to $X|_{\overline{\ell}}$ in this case.

Thus any implicant cube is guaranteed to either yield a conflict or to yield at least one implied assignment. In fact, any cube yielding at least one implied assignment must necessarily contain at least one implicant cube. Otherwise, every implied assignment would have to have at least one antecedent in the implication graph that is also an implied assignment, but since implied assignments cannot be sources in the graph, this is only possible if the graph has cycles (which is inconsistent with the graph's construction).

Besides their straightforward enumerability, cube implicants are interesting for the conclusions of their refutations. A refutation of a cube implicant allows its negation to be learned, but by definition, that negation subsumes the associated problem clause. Accordingly, the associated clause can be summarily strengthened. In this way, successive refutations of cube implicants gradually "deflate" the formula.

3.1.3 A note on validity

Both the density and the set of implicant cubes are, strictly speaking, associated with one specific set of clauses, and may change as clauses are learned or dropped. The set of implicant cubes is reasonably straightforward to update, since the effects of any update are limited to the cubes associated with the clause in question. On the other hand, the density of a cube may change whenever the variables that occur in its yielded set are involved. To avoid complex and potentially prohibitively expensive update computations, the present method makes the compromise of simply always using the score as it was first computed. Another alternative might be to periodically update the scores of only the top-scoring set of cubes.

3.1.4 Density distributions of industrial and random instances

If any benefit is to be gained by guiding search by cube densities, differences in density must necessarily capture meaningful differences in the practical solvability characteristics of the corresponding subproblems. Additionally, the advantages from using density as a difficulty measure must be at least partly such that they would not be reliably found by a CDCL-like search with a competent heuristic, otherwise the net effect is only to needlessly complicate the procedure. Inspecting the statistics and density distribution of implicants gives some insight regarding the first criterion; specifically, if the distribution tends to show at least some cubes being much denser than the mean, there is stronger evidence for the density being meaningfully descriptive than if the values only fell in a close range.

Figure 3.1 illustrates the density distributions of some of the industrial SAT instances contained in the 2008 SAT Race test set [1]. The following observations are immediately evident:

1. The mean density can vary greatly from one instance to the next. For some instances, it reaches the hundreds or thousands.

2. In many cases, a significant fraction of the implicants is refuted during the density-computing step, permitting a substantial number of clauses to be strengthened. In this way, the step also functions as a kind of preprocessing step.

3. Although it is rare for an implicant to be exceptionally dense, industrial instances commonly (though not universally) contain at least some

exceptionally dense implicants. The factor by which these are denser than the mean can vary quite widely.

By contrast, random instances appear to have none of these properties; analogous graphs for random 3-CNF instances are shown in Figure 3.2.

These preliminary observations are encouraging from the point of view of using cube density to guide search, but do not by themselves guarantee success. For example, computing the densities might be very costly for an instance with a high mean density, or refutations found during the computation may be easily navigated by the ordinary CDCL procedure. The ultimate usefulness of this approach must be assessed experimentally.

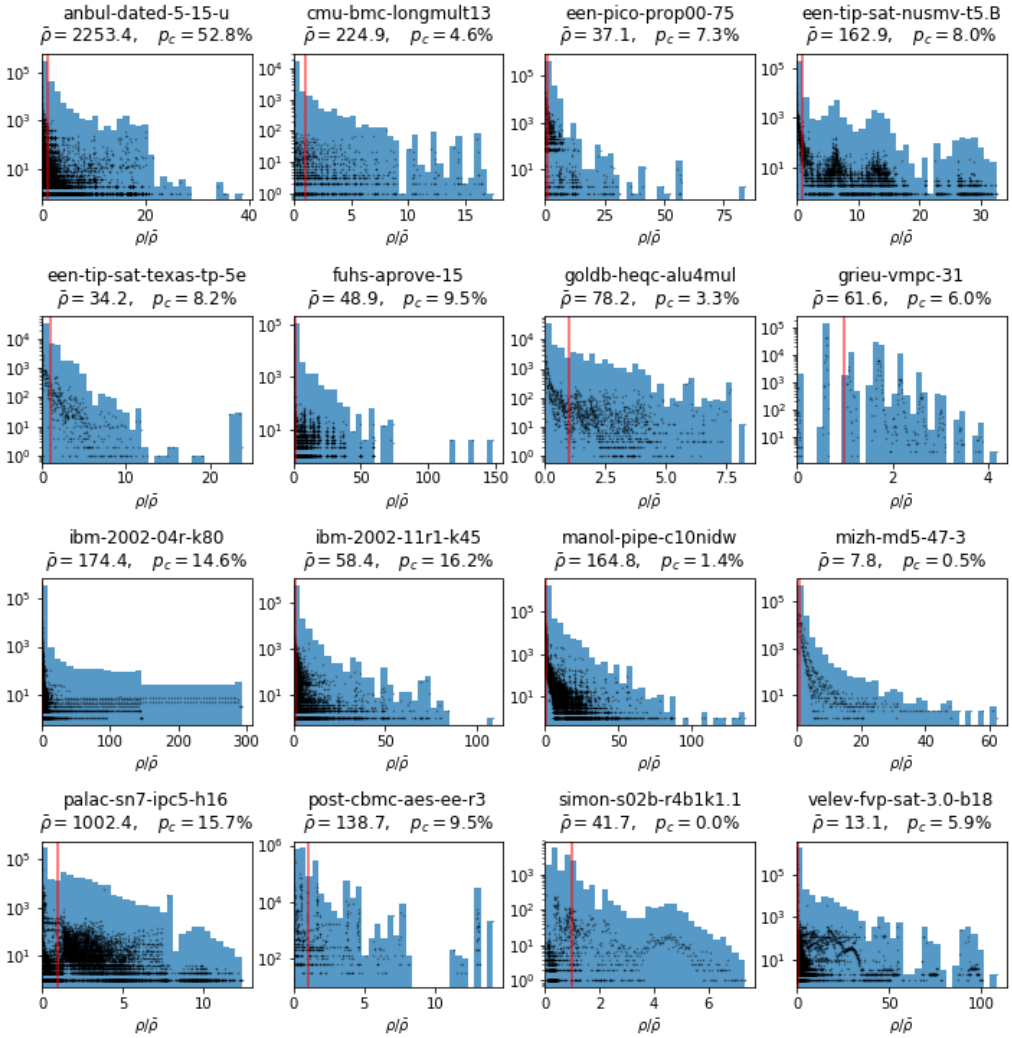


Figure 3.1: Distributions of implicant densities for a selection of industrial instances from the SAT-Race 2008 finals (main track). Implicants are found based on the formula closed under unit propagation and with satisfied clauses removed. Each implicant is counted once, regardless of how many clauses it is associated with. For easier comparison between instances, the normalized density $\rho/\bar{\rho}$ is used; the red vertical line indicates $\rho = \bar{\rho}$. Individual data points are overlaid on the histogram as a swarm of dots. Note the logarithmic scale on the vertical axis. The mean density $\bar{\rho}$ and the fraction of implicants refuted by unit propagation p_c are reported in the subplot titles under the instance identifier.

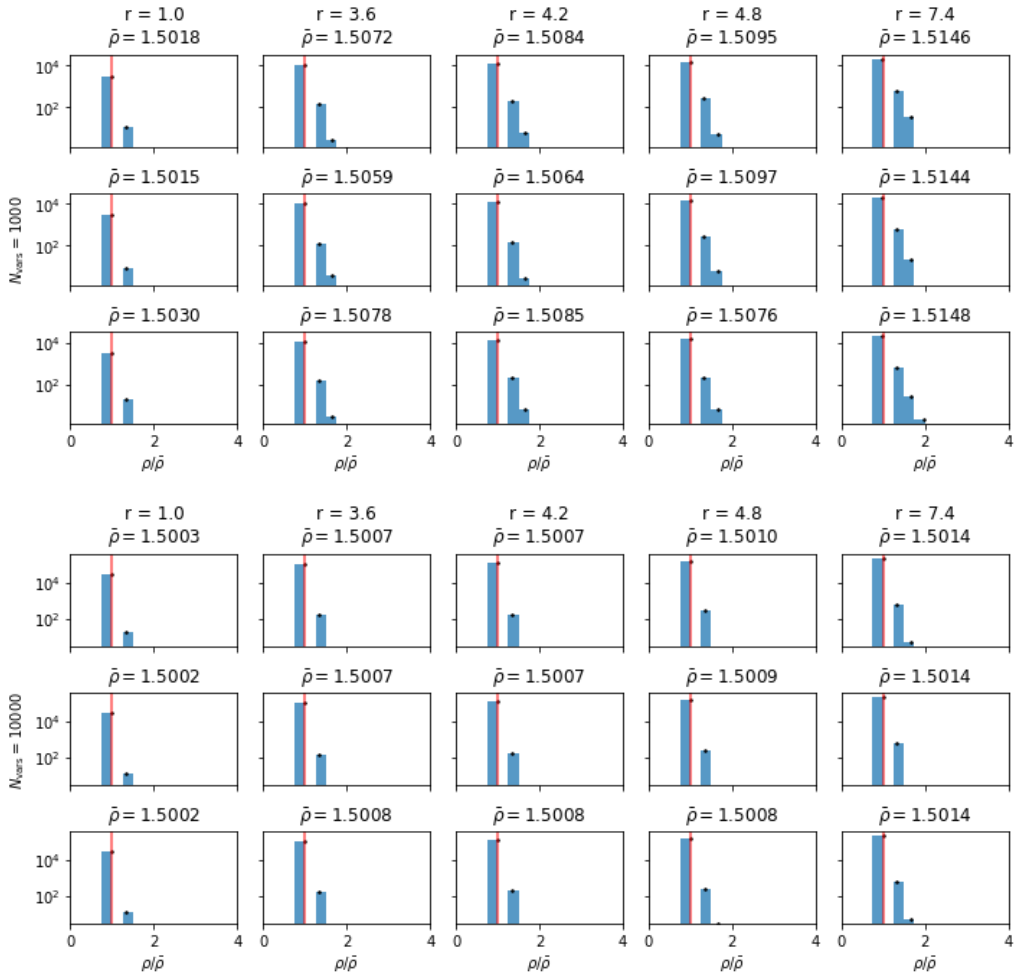


Figure 3.2: Distributions of implicant densities for random 3-CNF instances, with 1000 (above) or 10,000 (below) variables. Instances with different clause-to-variable ratios r are shown, three instances for each ratio. As with Figure 3.1, a red vertical line marks the mean implicant density. The mean density hardly reaches above the trivial value of $\bar{\rho} = 1.5$ for implicants associated with clauses of three variables. The number of refuted implicants is evidently negligible (none were encountered in the instances shown).

3.2 A solver guided by dense implicants

This section presents a complete solver that allocates part of its work to preferentially solving dense implicant cubes. It is implemented as an extension of MiniSat 2.1, specifically of its simp variant, which includes simplification of the formula by variable elimination as a preprocessing step [18]. For brevity, this original solver is simply referred to as MiniSat and the modified solver as Modified in the remainder of the text.

Figure 3.3 presents the differences between MiniSat and Modified in terms of their control graphs. In both cases, the main solving work occurs in a time-limited block, which is repeated while varying the limiting parameter according to a restart policy. In MiniSat, this block executes only CDCL search on the unconditioned problem, exiting when either the search reaches a determinate result or when an overall resource limit is reached. In Modified, the block has multiple steps. Besides the step for unconditioned CDCL search, there is a step to compute cube densities and a step for CDCL search conditioned on dense cubes found this way. The time limit is divided among the different steps, with roughly commensurate time allocated to each step.

The details and composition of the modified solver are developed below. Pseudocode sections are presented at the level of abstraction where the substance of the modifications occurs and treat the solver state (the list of problem clauses, the current search state, and so on) as implicitly accessible shared state. For clarity, the algorithm is presented without accounting for any overall resource limit; the checks for such a limit are not relevant to algorithm correctness, and their placement is quite self-evident.

3.2.1 CDCL search as the foundational solver mechanism

The solving procedure of MiniSat is shown in Algorithm 1. At its center is the CdclSearch(assumptions, budget) function, which is the solver's main workhorse in both MiniSat and Modified. It runs the DPLL-like core search procedure, with VSIDS heuristic and clause learning,[1] and can terminate in three ways:

1. The search discovers a satisfying assignment. The assignment is recorded as a model and the function returns sat.

[1]This corresponds to the C++ function `Minisat::Solver::search(int)`, with an appropriate value of `Minisat::Solver::assumptions`.

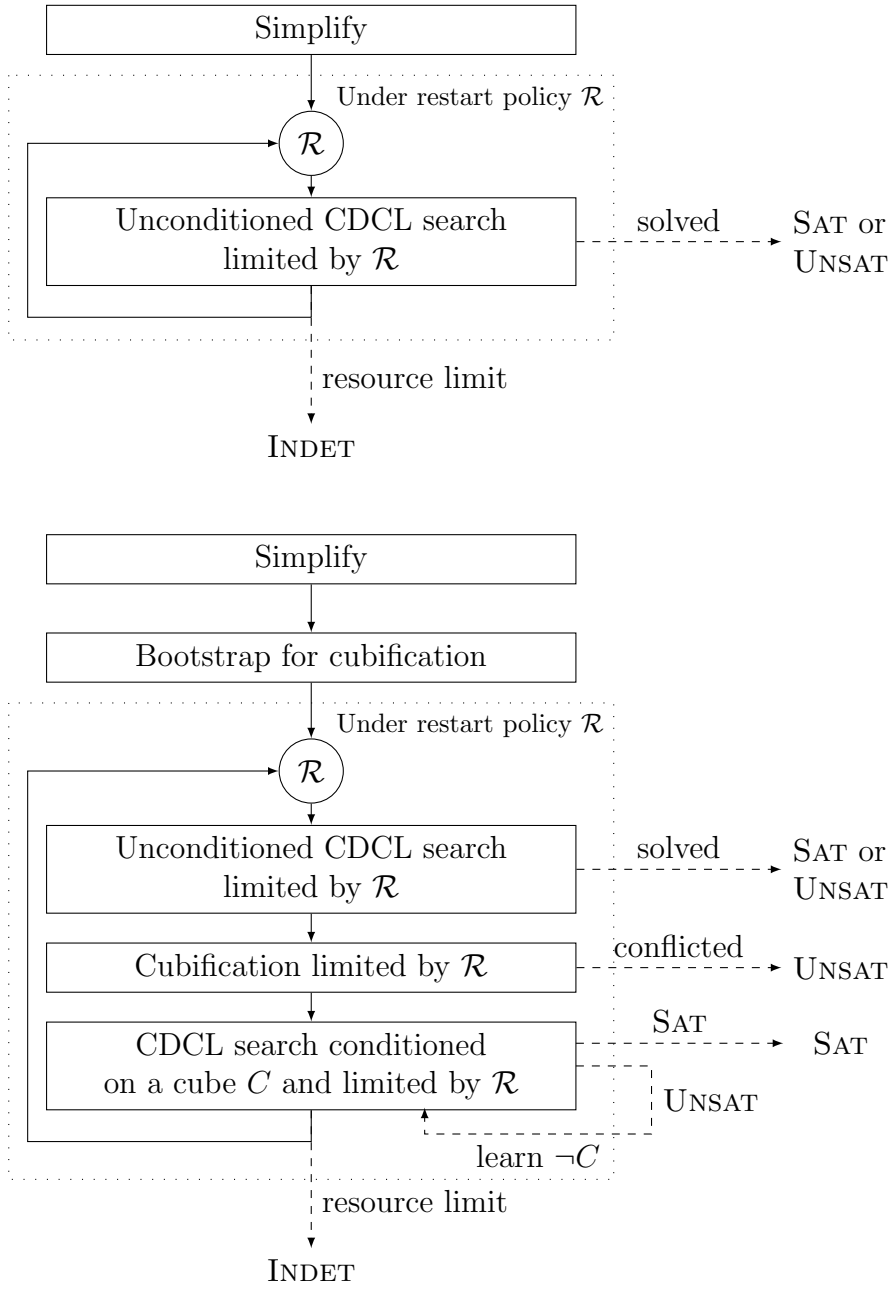


Figure 3.3: High-level differences between MiniSat (above) and Modified (below), presented in terms of their control graphs. The node $\mathcal{R}$ represents an update of the iteration budget.

Algorithm 1 Solving procedure of MiniSat

```
1: procedure MinisatSolve()
2:     ret ← indet
3:     UnitPropagate()
4:     EliminateVariables()
5:     while ret = indet do
6:         n ← NextRestartInterval()
7:         ret ← CdclSearch(∅, n)
8:     end while
9:     return ret
10: end procedure
```

2. The search derives a refutation under the assumptions. The final conflict clause is recorded and the function returns UNSAT.

3. The search exhausts its conflict budget without a determinate conclusion and aborts. The function returns INDET.

In the case of MiniSat, after initial simplification is completed, CdclSearch is simply repeatedly called with an empty set of assumptions and a conflict budget dictated by the current restart policy. The loop is broken once the return value indicates that a conclusion has been reached.

3.2.2 Scoring implicants by cubification

In Modified, search steps conditioned on specific dense implicant cubes alternate with steps of unconditioned search. The densities of the implicants must therefore also be computed. Since going through all the implicants can add up to a significant amount of work, sometimes much more than solving the problem directly through unconditioned search, the computation is divided into parts and interleaved with search steps. The work within each part is also optimized so as to avoid a modest amount of unnecessary work. The atomic component of a density-computing step is to take a problem clause and to visit all of its associated implicant cubes, a procedure here referred to as *cubifying* the clause.

The cubification procedure is based on making and undoing individual decision assignments, along with their consequences under unit propagation, then inspecting the search state that follows. The terminology of stack operations is used here to speak of the sequence of decision assignments and their cancellations, since this reflects the way that these operations must necessarily be ordered: *pushing* a decision assignment means making that decision on

a new decision level and running unit propagation, while *popping* a decision assignment means undoing the most recently pushed decision assignment and all of its consequences. To compute the density of a given cube, one simply successively pushes the decision assignments that correspond to its literals, skipping any literals redundant with the already-known assignments, then compares the resulting number of assignments to the number of assignments when the stack was empty and divides the difference by the size of the stack.

The cubification procedure of MODIFIED for a clause X is shown in Algorithm 2. The procedure loops over the implicants $\neg(X|_{S[i]})$ associated with X, processing only those implicants which have not yet been scored. (The order of the implicants S, which is also the order in which assignments are pushed, is specified by a procedure described in more detail below.) As a loop invariant, when processing of the ith implicant starts, the decision stack is a subset of $\neg(X|_{S[i]})$ in the order given by S. The decision assignments that remain to complete the implicant are pushed and the density of the implicant, as well as that of all intermediate cubes which yield any implied assignments, is recorded (lines 6-26). If a contradiction is derived at any point, cubification terminates and the clause is strengthened. Processing of an implicant ends by popping decision assignments (lines 27-31). Popping only continues up to the point required to maintain the loop invariant, allowing shared decision levels to be reused; the resulting traversal is illustrated in Figure 3.4. Finally, the stack is cleared, placing the solver back at the root of the decision tree.

A further optimization technique targets the order S in which decision assignments are pushed. Implicants that occur earlier in the search order require more work to reach, therefore any decision assignments whose corresponding implicants have already been scored (and which can, therefore, be skipped) are placed first in the variable order. This avoids the greatest possible amount of work. Among the remaining decision assignments, whose corresponding implicants do need to be scored, ones occurring earlier need to be pushed fewer times than those occurring later. These assignments are therefore placed in descending order of the expected work involved in pushing them, estimated by the number of implied assignments yielded when they are the first decision assignment pushed onto the stack. The procedure is shown in Algorithm 3.

These two procedures make use of some data structures that are new in MODIFIED:

1. An associative container W, which maps each problem literal to its expected pushing work. The initial value is ∞ for all literals.

2. A queue of clause references Q_c, indicating clauses awaiting cubifica-

Algorithm 2 Score the implicants associated with a clause X

```
1: function CUBIFY(X)
2:     S ← MAKEPUSHORDER(X)
3:     for i = 1, ..., |S| do
4:         C ← ¬(X|_{S[i]})
5:         if ρ[C] is not defined then          ▷ if C has not yet been scored
6:             for j = 1, ..., |S| do
7:                 if j ≠ i and S[j] is not a current assignment then
8:                     PUSHDECISIONASSIGNMENT(S[j])
9:                     if a contradiction has been derived then
10:                        C' ← current decision assignments as a cube
11:                        delete problem clause X
12:                        add ¬C' to problem clauses
13:                        enqueue ¬C' in Q_c
14:                        POPALLDECISIONASSIGNMENTS()
15:                        return
16:                    else if S[j] is the sole decision assignment then
17:                        W[S[j]] ← propagations during last push
18:                    end if
19:                end if
20:                if current decisions yield implied assignments then
21:                    C' ← current decision assignments as a cube
22:                    ρ[C'] ← current yield / |C'|
23:                    add X to P[C']
24:                    enqueue C' in Q_s with value ρ[C']
25:                end if
26:            end for
27:            for j = |S|, ..., i do
28:                if S[j] is on the decision stack then
29:                    POPDECISIONASSIGNMENT()
30:                end if
31:            end for
32:        end if
33:    end for
34:    POPALLDECISIONASSIGNMENTS()
35: end function
```

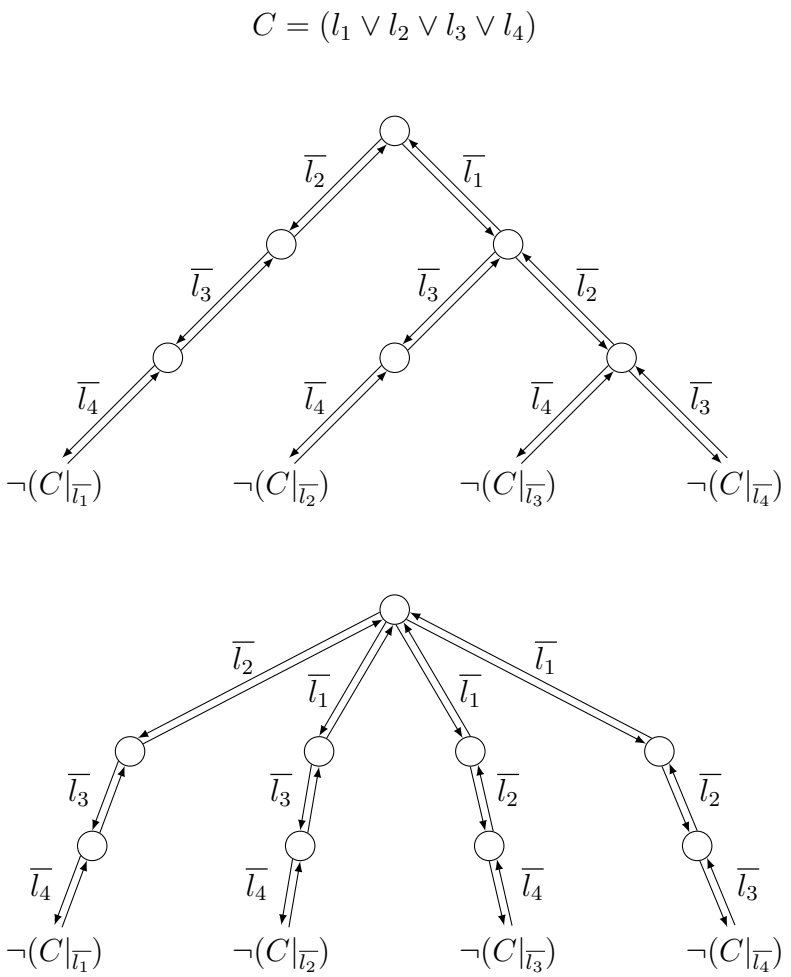


Figure 3.4: An economical traversal of the decision tree (above) enumerating all implicant cubes associated with a given problem clause, here illustrated for a clause C of four literals. Traversal proceeds counter-clockwise from the root, descending arrows indicating that a decision assignment is pushed, ascending arrows that it is popped. This method avoids a number of unnecessary changes of propagation level compared with the naive approach (below). For a cube of N literals, economical traversal requires $\frac{N(N+1)}{2} - 1$ push and pop steps, compared to $N(N-1)$ push and pop steps in the naive case.

Algorithm 3 Determining the cubification push order for a clause X

```
1: function MakePushOrder(X)
2:     S_0 ← empty sequence
3:     S_1 ← empty sequence
4:     for i = 1, ..., |X| do
5:         ℓ ← ¬X[i]
6:         C ← ¬(X|_ℓ)
7:         if ρ[C] is defined then          ▷ if C has already been scored
8:             append ℓ to S_0
9:         else
10:            add X to P[C]
11:            append ℓ to S_1
12:        end if
13:    end for
14:    sort S_1 by values in W in descending order
15:    return Concatenate(S_0, S_1)
16: end function
```

tion.

3. A priority queue of cubes Q_s, in descending order of density, indicating which cube is next for conditioned search.

4. An associative container ρ, which maps each implicant cube to its density. Additionally, a cached value is maintained for the sum of densities $\sum_C \rho[C]$, such that the mean density $\bar{\rho} = \frac{\sum_C \rho[C]}{|\rho|}$ is always accessible.

5. An associative container P, which maps each implicant cube to a set of clause references with which it is known to be associated (i.e., those clauses that are subsumed by the negation of that implicant cube).

These data structures, like the solver state, are shared components of state and persist throughout execution. The clause references should be adequate for efficient deletion or replacement of the clause and should be resilient to changes to the clause set, including the deletion of the clause. A mechanism accomplishing this is described in Section 3.2.5.

In addition to the implicants directly associated with the clause being cubified, some of their subcubes may themselves be (or contain) implicants. Such cases are identified by them yielding implied assignments, and their scores are immediately recorded. If at any point during the procedure a cube causes a conflict, the clause under inspection is immediately strengthened

to the negation of the conflicting cube, the strengthened clause is enqueued for cubification, all decisions are popped, and the procedure exits. Note that the negation of a cube yielding a conflict is necessarily an implicate of the formula and that the negation of any decision cube reachable via the procedure necessarily subsumes the original clause.

3.2.3 CDCL search conditioned on a cube

For the case of search conditioned on a cube, CdclSearch is called with that cube as the set of assumptions A. The interpretation of the return value is then as follows:

- If CdclSearch returns SAT, a model M exists under the assumptions A. It follows that $M \cup A$ is a model in the unconstrained case, and the full problem is SAT.
- If CdclSearch returns UNSAT, some set of literals $A' \subseteq A$ is inconsistent with the formula, and the clausal implicate $\neg A'$ is obtained. The final conflict clause recorded by CdclSearch is an implicate of this kind, and a minimal one in the sense that no further removal of literals is justified by the analysis at hand. (If the final conflict clause is empty, search found a refutation for the unconstrained formula, and the full problem is shown to be UNSAT.)

It follows that when CdclSearch conditioned on A derives a conflict, the final conflict clause $\neg A'$ can be learned. In principle, this is analogous to regular clause learning. However, clauses learnt by regular CDCL tend to be large (i.e., weak) and to necessarily increase the total number of clauses. Due to its origin, $\neg A$ is of comparatively limited size (i.e., strong), and since it subsumes and therefore can replace the clause associated with the implicant A, it typically does not increase the number of clauses (unless the associated clause has previously been deleted elsewhere).

Algorithm 4 expresses the conditioned search and the handling of its consequences as the function CubeSearch(cube, budget). Its return value is the conclusion that was reached for the full problem. Unlike clauses learnt by regular CDCL, $\neg A'$ is added as a clause of the main formula (i.e., not as a learnt clause subject to clause database culling), for the reasons outlined in the preceding paragraph.

3.2.4 Composition of Modified

Algorithm 5 shows the composition of Modified in terms of the functions CdclSearch, Cubify (Algorithm 2), and CubeSearch (Algorithm 4).

Algorithm 4 Search conditioned on a cube

```
1: function CubeSearch(C, n)
2:     ret ← CdclSearch(C, n)
3:     if ret = INDET then
4:         return INDET
5:     else if ret = SAT then
6:         return SAT
7:     else
8:         X ← final conflict clause
9:         if X is empty then
10:            return UNSAT
11:        else
12:            for each clause Y in P[C] do
13:                if the problem clause Y still exists then
14:                    remove Y from problem clauses
15:                end if
16:            end for
17:            add X to problem clauses
18:            enqueue X in Q_c
19:            remove C from Q_s
20:            erase key C from P
21:            erase key C from ρ
22:            return INDET
23:        end if
24:    end if
25: end function
```

Work is split among the different kinds of search and cubification with the intent of allocating roughly equal effort to each step. Like in MiniSat, unconditioned search uses the restart interval directly as a limit on the number of conflicts. Search conditioned on a cube, being of a similar character, also takes the conflict limit interpretation, with the difference that it allows the conflicts to occur across multiple search calls. This is to prevent conditioned search from getting effectively starved due to a sequence of easily refuted cubes. Cubification does not generally hit conflicts, so instead the number of propagations during CdclSearch is recorded and a proportional limit on the number of propagations is imposed on a sequence of Cubify calls.

Modified is parameterized as follows:

1. k_c is the constant of proportionality between propagations during search and propagations during cubification. It allows for adjustments to the time spent in cubification, such as if the work per propagation is found to be different between the top of the search tree and the deeper levels.

2. $w_{\max}$ is the maximum width for a clause to be cubified. It allows the algorithm to avoid the largest clauses, which are associated with comparatively great cubification work and the refutation of whose implicants derives comparatively weak clausal implicates.

3. k_t controls the relative density required for a cube to be considered worth searching. Search conditioned on a cube is only executed if a cube at least k_t times as dense as the average implicant is available.

In addition to the above parameters, the normal parameters to MiniSat apply.

3.2.5 Practical implementation details

The preceding sections capture the essential logic of Modified. This final section addresses some details about the implementation that do not affect the logic but are presented for the sake of completeness.

First, an index of the problem clauses (i.e., those not subject to clause database culling) is maintained as a set of clauses. The index is first populated after the call to EliminateVariables and is updated each time a clause is added, either in Cubify when a summary subsumption is found or in CubeSearch when a cube is refuted. The primary purpose of this index is to prevent duplication of clauses.

Second, a data structure is used to associate each problem clause with a unique, persistent identifier, which is used as the clause reference in the Q_c

Algorithm 5 Solving procedure of MODIFIED

```
1: procedure ModifiedSolve()
2:     ret ← INDET
3:     UnitPropagate()
4:     EliminateVariables()
5:     enqueue current clauses for cubification
6:     while ret = INDET do
7:         N_c ← NextRestartInterval()
8:         ret ← CdclSearch(∅, N_c)
9:         if ret ≠ INDET then
10:            return ret
11:        end if
12:        N_p ← number of propagations during search step
13:        n ← 0
14:        while n < k_c × N_p and Q_c is not empty do
15:            X ← pop a clause from Q_c
16:            if |X| ≤ w_max then
17:                Cubify(X)
18:                if a conflict has been derived then
19:                    return UNSAT
20:                end if
21:                n ← n + number of propagations during CdclSearch
22:            end if
23:        end while
24:        n ← 0
25:        while n < N_c and Q_s is not empty do
26:            c ← densest cube in Q_s      ▷ random choice if there are many
27:            if ρ[c] < k_t × ρ̄ then
28:                break
29:            else
30:                ret ← CubeSearch(c, N_c − n)
31:                if ret ≠ INDET then
32:                    return ret
33:                end if
34:                n ← n + number of conflicts during CdclSearch
35:            end if
36:        end while
37:        delete satisfied clauses
38:    end while
39:    return ret
40: end procedure
```

and P data structures. MiniSat keeps track of its problem clauses in a list so that, at any given moment, an index into the list suffices to locate a clause. However, the momentary indices into the list can change when clauses are added or removed. A bidirectional mapping is used to associate this transient clause index to a persistent identifier and vice versa. The index is updated whenever a clause is added or removed and is naturally able to tell when a clause with a given persistent identifier has since been deleted. The MiniSat procedure for removing satisfied clauses is modified to maintain the validity of the index.

Chapter 4

Measurements

The performance of Modified in comparison to MiniSat was evaluated by applying both solvers to a set of benchmark instances, then comparing the number and running time of the successful executions between the two. The 100 instances used in the Main Track finals of the SAT-Race 2008 competition [1] were used as the benchmark set. This set was chosen because it represents a wide range of problem sizes and fields of application, is relatively compact, and was composed with the competition's 15-minute time limit in mind. These properties make it more convenient to work with than some other competition sets.

All computations were run on workstations with 3.40 GHz Xeon E3-1230 v5 CPUs and 16 gigabytes of memory. The parameters specific to Modified[1] were chosen as follows:

- $k_c = 1$, corresponding to the default assumption that unit propagation during the search step and unit propagation during the cubification step take roughly the same amount of time;
- $w_{\max} = 6$, reflecting an observation that the densest implicants appear to predominantly be quite short; and
- $k_t = 5$, based on a visual inspection of the implicant density distributions of industrial instances.

This combination emerged as a reasonable choice among a selection of different candidates for which test runs were performed. All other parameters, except for the random seed, retained the default values chosen in MiniSat. As in the original competition, a time limit of 900 seconds (15 minutes)

[1] Described in Section 3.2.4.

	original form			permuted form		
solved by	SAT	UNSAT	INDET	SAT	UNSAT	INDET
both solvers	420	413	120	380	414	123
MiniSat only	10	17	20	32	14	37
Modified only	9	11	27	27	10	46

Table 4.1: Solved instance counts for each solver.

was imposed. The wall-clock time from launching the solver process to its termination was taken to be the running time.

Each instance was solved 40 times in total: 10 times per solver in its original form and 10 times per solver in permuted form. Permuted forms were created by randomizing the order of clauses, the labeling of the variables, and the variable polarities, resulting in an equisatisfiable formula. For a given instance and form, each pair of runs (one of MiniSat, one of Modified) used a unique random seed for the solver (given by the `-rnd-seed` command line parameter).

4.1 Results

Table 4.1 shows the number of solved instances by form and solver outcome. Cactus plots of the running times are shown in Figure 4.1. Though each of the solvers was capable of solving some instances that the other was not, over the entire benchmark set, MiniSat was somewhat more effective than Modified. A similar result is seen in Table 4.2, which lists on how many instances each of the solvers was faster than the other. The five largest instance families contained in the set, comprising 10 instances each (20 in the case of `ibm`), are considered both separately and as part of the full set, with the intention of exposing solvability characteristics that may be specific to a particular kind of instance. Overall, out of the instances solved by at least one of the solvers, Modified was faster on 35% of original-form instances (29% of permuted-form instances). In other words, the baseline solver remained faster on average. This advantage was not universal across all classes of instances, however: Modified was faster on the satisfiable instances from the `ibm` and `mizh` benchmark families. Broadly, the advantage enjoyed by MiniSat appeared to be greater on the unsatisfiable than the satisfiable instances and on the permuted than the original form.

Among those pairs of runs where both solvers reached a determinate result, the ratio of the running times varied considerably. By way of quan-

benchmark family		original form N_{MIN}	original form N_{MOD}	original form p	permuted form N_{MIN}	permuted form N_{MOD}	permuted form p
ibm	SAT	61	89	59%	95	54	36%
	UNSAT	49	1	2%	46	4	8%
post	SAT	19	1	5%	19	1	5%
	UNSAT	43	27	39%	57	13	19%
manol	UNSAT	67	33	33%	78	19	20%
velev	SAT	20	20	50%	28	12	30%
	UNSAT	16	5	24%	13	8	38%
mizh	SAT	39	60	61%	46	49	52%
misc.	SAT	122	8	6%	99	36	27%
	UNSAT	133	67	34%	141	59	30%
overall	SAT	261	178	41%	287	152	35%
	UNSAT	308	133	30%	335	103	24%

Table 4.2: Solver comparison for instances solved by at least one solver. N_{MIN} indicates the number of instances on which MINISAT was faster, N_{MOD} the number on which MODIFIED was faster and $p = N_{\text{MOD}}/(N_{\text{MIN}} + N_{\text{MOD}})$.

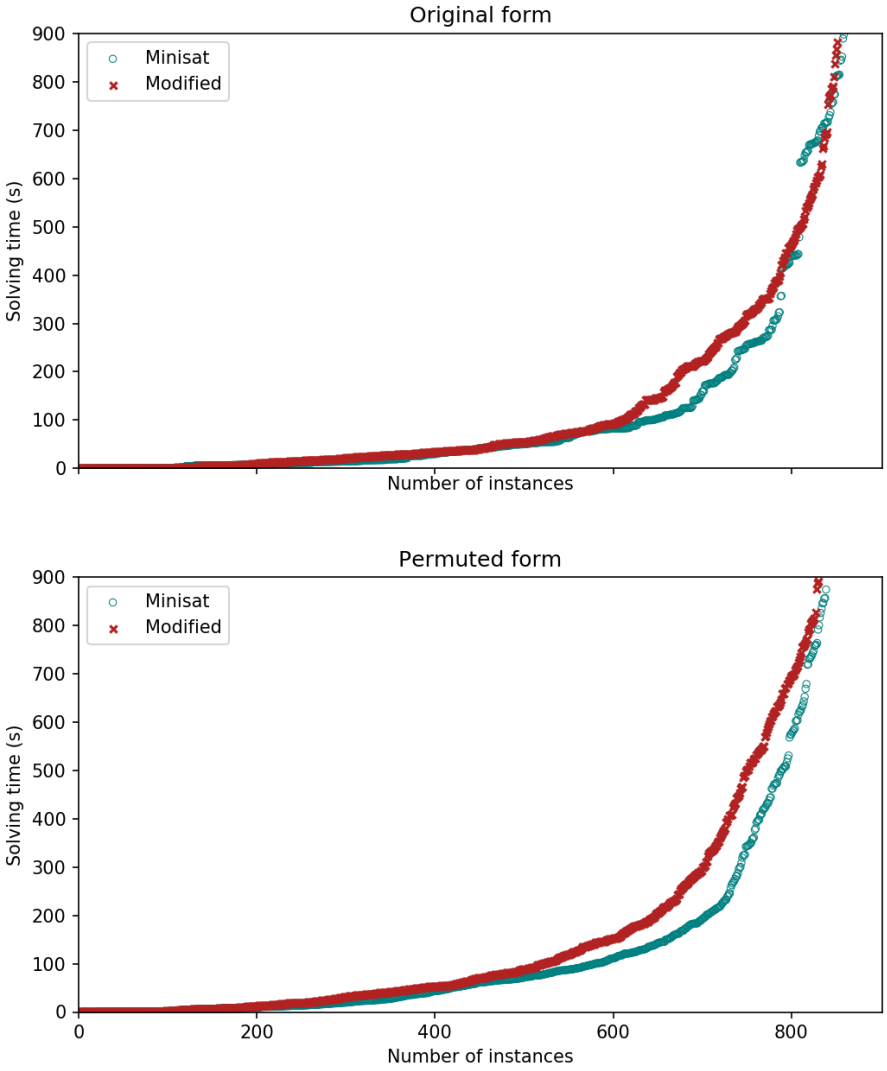


Figure 4.1: Runtime comparison between MiniSat and Modified. Each of the 100 instances occurs 10 times per solver and instance form, adding up to a total of 1000 solving attempts each.

family	outcome	original form N	original form $\hat{\mu}$	original form $\hat{\sigma}$	permuted form N	permuted form $\hat{\mu}$	permuted form $\hat{\sigma}$
ibm	SAT	150	0.213	1.035	144	-0.107	0.897
	UNSAT	50	-0.682	0.571	50	-0.540	0.458
mizh	SAT	90	0.293	1.439	75	-0.076	1.411
velev	SAT	40	0.011	0.383	34	-0.045	0.647
	UNSAT	15	-0.233	0.388	13	-0.471	0.251
manol	UNSAT	95	-0.205	0.387	91	-0.427	0.590
post	SAT	20	-0.342	0.208	20	-0.230	0.175
	UNSAT	63	-0.161	0.440	61	-0.241	0.362
misc.	SAT	120	-0.756	1.129	107	-0.523	0.992
	UNSAT	190	-0.226	0.583	199	-0.147	0.522
overall	SAT	420	-0.093	1.184	380	-0.219	1.025
	UNSAT	413	-0.267	0.538	414	-0.280	0.526

Table 4.3: Statistical properties of the log ratio A.

titative analysis, one can consider the statistical properties of the log ratio,

$$A(T_0, T_1) = \log \frac{T_0}{T_1}, \tag{4.1}$$

where T_0 and T_1 are the solving times of MiniSat and Modified, respectively. (The logarithm is to make the value symmetric with respect to the order of the solvers, up to a sign change; a positive value indicates that the latter solver is faster and vice versa.) The resulting statistics of the log ratio are laid out in Table 4.3. As was the case with the results shown in Table 4.2, `MiniSat` was faster overall, but on the satisfiable instances of the `ibm` and `mizh` families, `Modified` was faster. The comparison of running times is also visualized in Figures 4.3 and 4.4 (overall) and in Figures 4.5 and 4.6 (by family).

During runs of Modified, the time spent in each major step was recorded, allowing the fraction of time spent in cubification to be obtained. Figure 4.2 displays these results across the original-form instances and contrasts them with the log ratio A in a two-dimensional histogram. Perhaps surprisingly, no correlation between the time spent in cubification and the relative solving performance is apparent from this picture of the distribution.

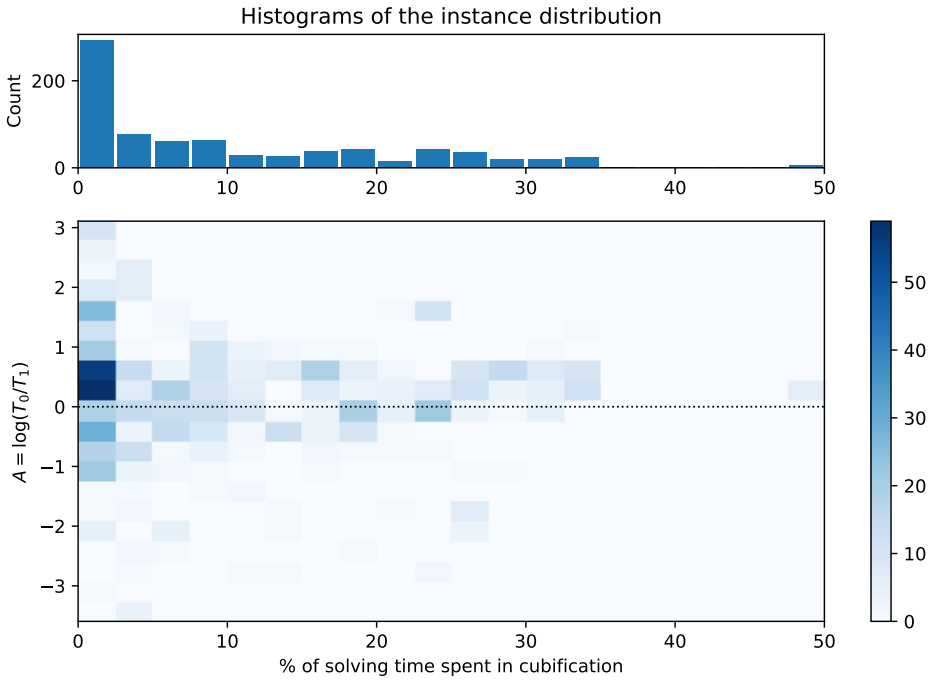


Figure 4.2: Distribution of original-form instances across both the fraction of time spent in cubification and the log ratio of the solving times. Only instances for which both solvers reached a determinate result and in which MODIFIED spent a nonzero time in cubification are included.

4.2 Discussion

There are several challenges involved in making a fair comparison between MiniSat and Modified. This section briefly mentions some possible confounding effects.

In Modified, solving progress is primarily made in the unconditioned and the cube-conditioned search steps. Both steps use the same shared state, comprising the set of learnt clauses, the VSIDS heuristic values, the pseudo-random number generator state, etc. so that the updates made by one step also affect the other step. In principle, both steps make updates of the same kind, so that in the ideal case, they are complementary to each other. However, especially the VSIDS heuristics and the learnt clauses can be quite local in their usefulness, so that the updates from one step may actually thwart the progress of the other by degrading the heuristic values and culling the learnt clauses that are only significant in the subspace seen by the other step. In other words, a situation of competing search regions is possible. There is therefore also a reasonable argument for making parts of the shared state either partially or completely step specific. In Modified, the choice is made to keep the state fully shared, as this way the changes to the core code of MiniSat are minimized.

It is also possible that the data structures introduced in Modified, although chosen with performance in mind, are under-engineered in comparison to the specialized containers used in MiniSat. The comparatively worse memory optimization may be causing a relative slowdown. Memory considerations may also explain why both solvers do worse on the permuted-form instances. A permutation may scatter related clauses and variables that may have cohered in the original form as a consequence of how it was constructed. The loss of performance from permutation is greater for Modified than for MiniSat, which also supports the idea that Modified could benefit from more memory optimization.

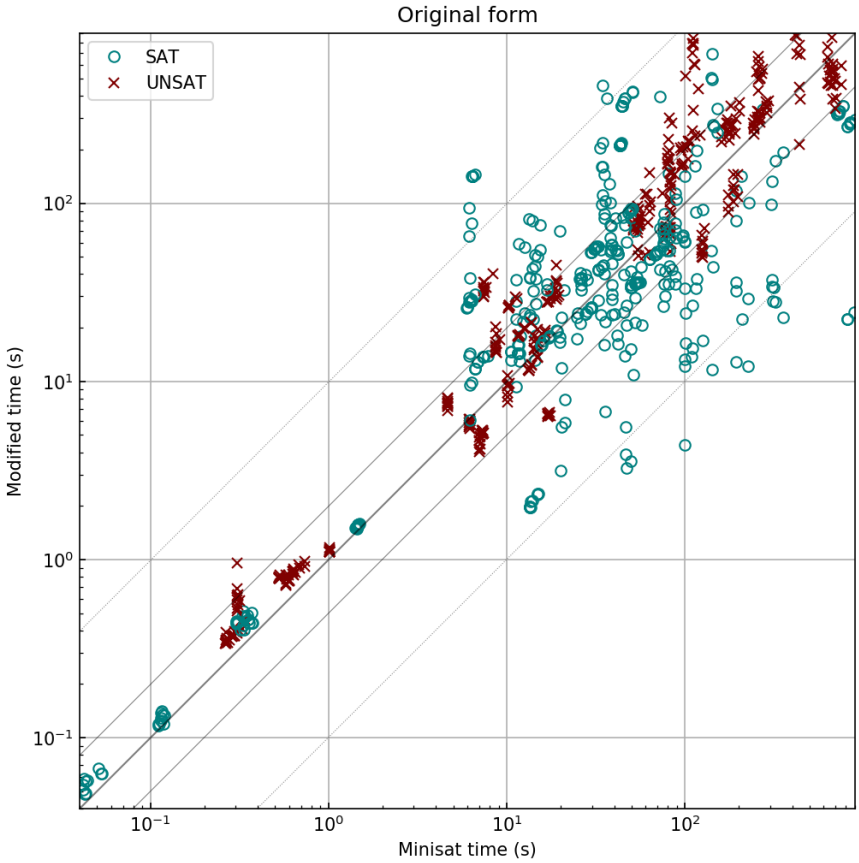


Figure 4.3: Solving time comparison for those pairs of original-form instance and random seed that were solved by both MiniSat and Modified. The central diagonal marks where the running times are equal. The outer diagonals mark the thresholds of 2:1 and 10:1 solving time ratios.

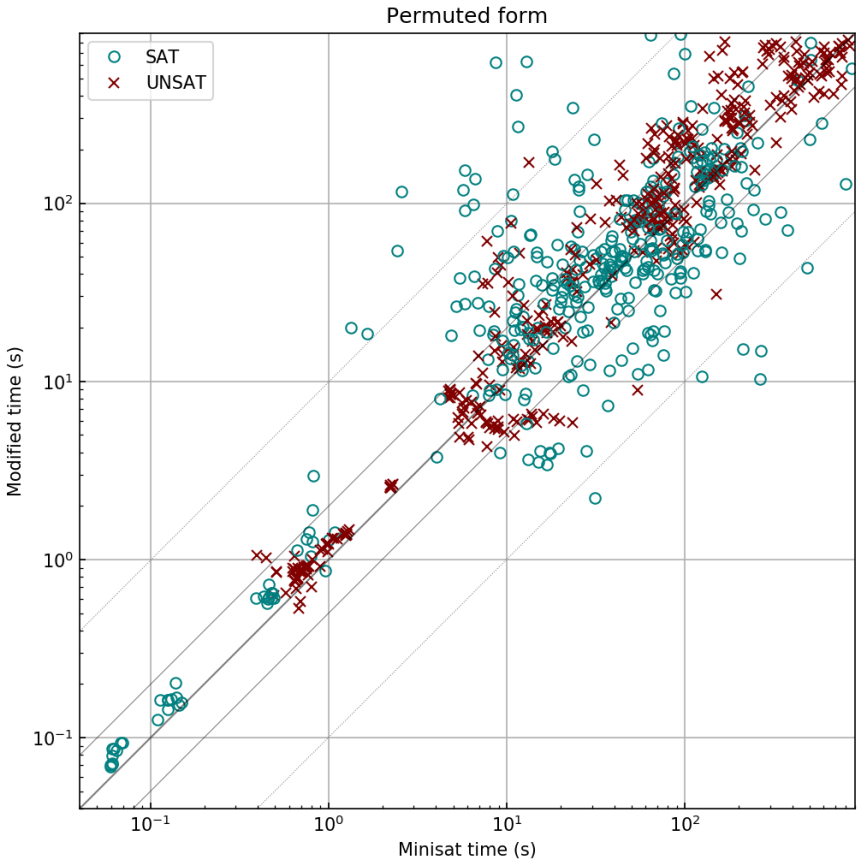


Figure 4.4: Solving time comparison for those pairs of permuted-form instance and random seed that were solved by both MiniSat and Modified. The central diagonal marks where the running times are equal. The outer diagonals mark the thresholds of 2:1 and 10:1 solving time ratios.

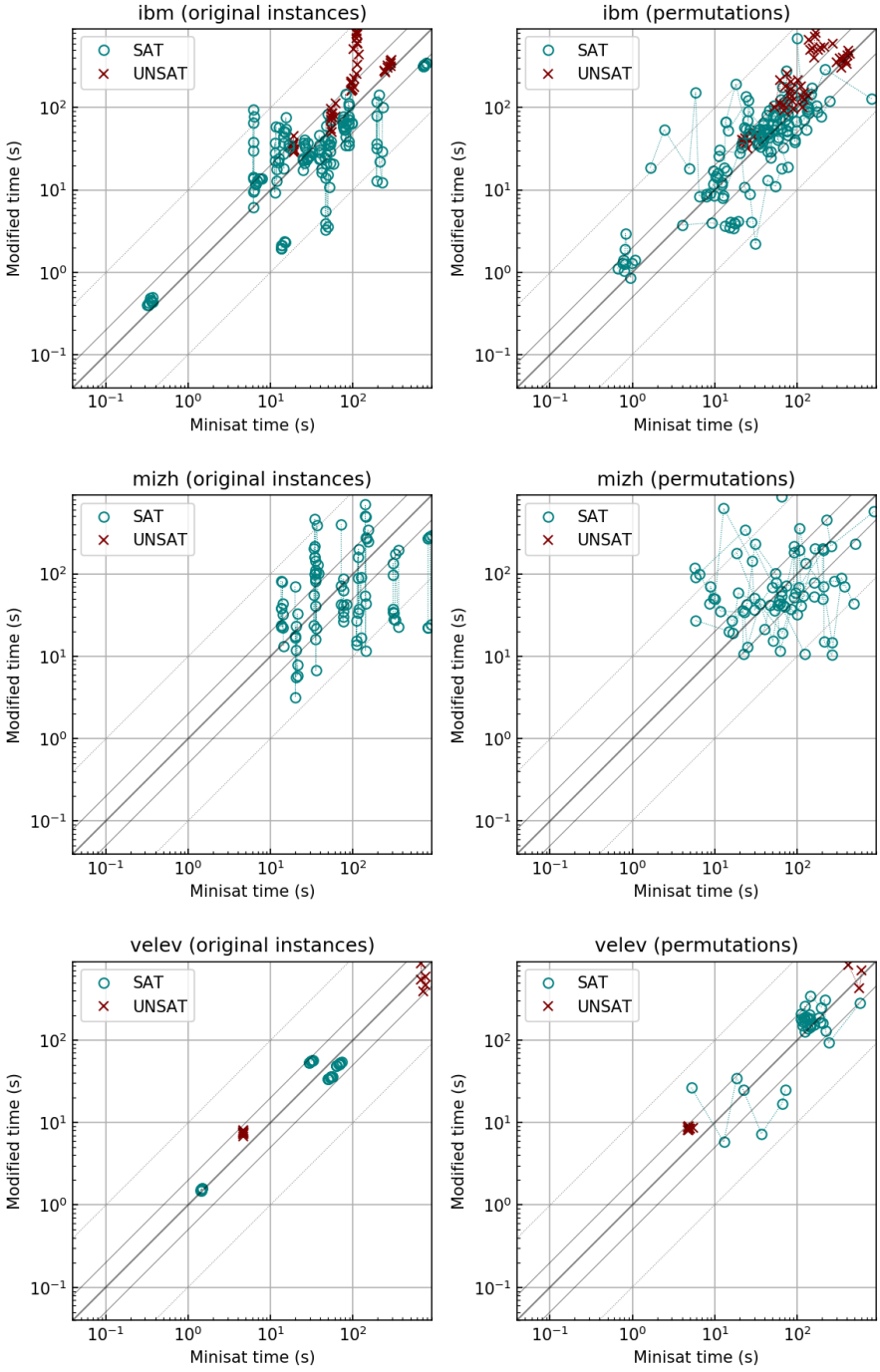


Figure 4.5: Solving time comparison for instance families `ibm`, `mizh` and `velev`.

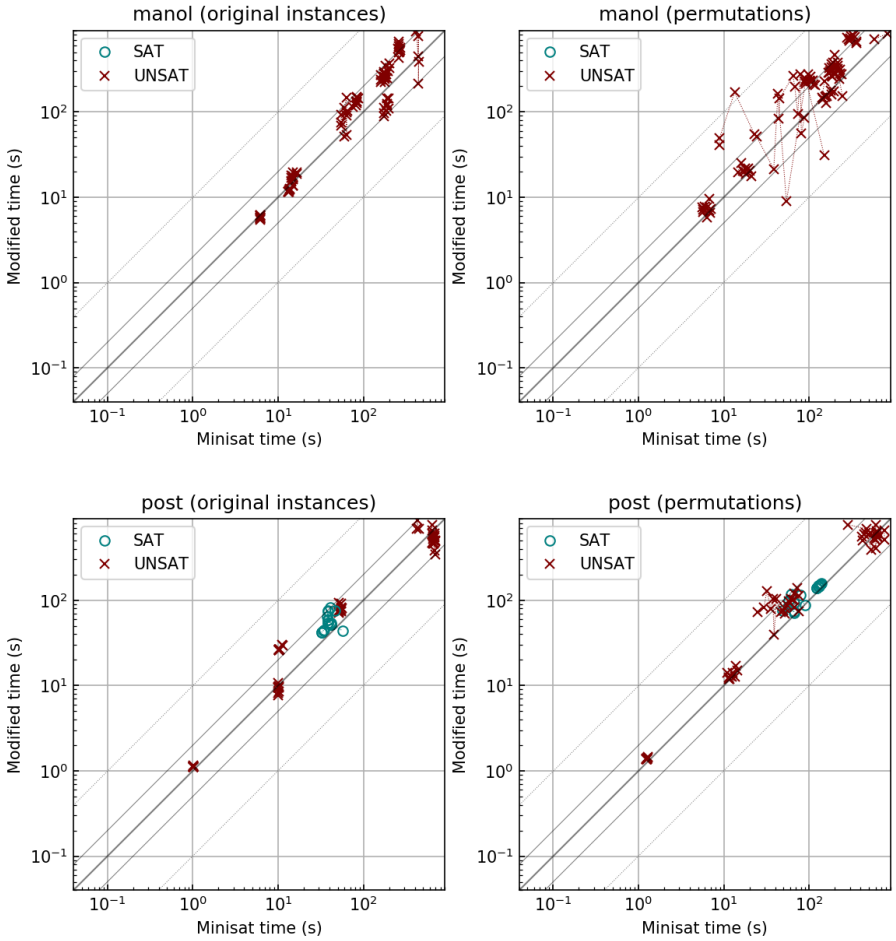


Figure 4.6: Solving time comparison for instance families `manol` and `pipe`.

Chapter 5

Conclusions

This thesis proposes and evaluates a solving technique for use in CDCL-based SAT solvers, namely the use of the implicative density as a heuristic to find regions of the search space that appear comparatively easy to refute and to preferentially target these regions during search. This technique was implemented as a modification to the MiniSat 2.1 solver.[1] The time taken by the modified solver to solve a range of benchmark instances was compared to the time taken by the original solver.

Across the full set of benchmarks, the modified solver was faster on 35% of non-permuted and 29% of permuted instances, considering those instances that were solved by at least one of the solvers. Among those instances solved by both solvers, the mean log ratio of the running times of the solvers also indicated an overall advantage for the original solver. However, the modified solver did do somewhat better by this measure on the satisfiable instances in the `ibm` and `mizh` benchmark families.

Although these results do not support the use of this technique in general and in the form implemented and studied in this thesis, further development could still find merit in the underlying ideas. The results seen on the satisfiable `ibm` and `mizh` instances suggest that instances with particular structural characteristics might benefit from the procedure as implemented. The implicant density distribution appears to capture something of the characteristics of an industrial SAT instance, indicating that the implicant density could be useful in other contexts, and since the density-computing step appears to often uncover a substantial number of subsumptions, an implicant-based preprocessing step could be useful.

Apart from finding alternative applications for the implicant density, it might be possible to make the present algorithm viable through improve-

[1] The implementation source code is available online at `https://github.com/lippinj/minisat-cubing`.

ments to the design or optimization of the implementation. For instance, one could run the cubification and cube-conditioned search steps in parallel with a CDCL solver, possibly across multiple threads. These steps would function as a kind of subsumption generator, which the main solver process can query at the start of each restart cycle for changes to its formula. On the other hand, if the solver is kept sequential, memory optimization might improve the results, as might reducing the degree to which the unconditioned and cube-conditioned search steps share solver state.

Bibliography

[1] SAT-Race 2008, 2008 (accessed October 2, 2019). URL `https://baldur.iti.kit.edu/sat-race-2008/index.html`.

[2] The International SAT Competition, 2019 (accessed April 13, 2019). URL `http://www.satcompetition.org`.

[3] Dimitris Achlioptas. Random satisfiability. In *Handbook of Satisfiability*, pages 245–270. IOS Press, 2009.

[4] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. On the structure of industrial SAT instances. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 127–141. Springer, 2009.

[5] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Towards industrial-like random SAT instances. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 387–392, 2009.

[6] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 410–423. Springer, 2012.

[7] Carlos Ansótegui, María Luisa Bonet, Jesús Giráldez-Cru, and Jordi Levy. The fractal dimension of SAT formulas. In *Automated Reasoning: 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, Vienna, Austria, July 19-22, 2014, Proceedings*, pages 107–121. Springer, 2014.

[8] Gilles Audemard and Laurent Simon. Glucose and Syrup in the SAT Race 2015. In *Reports on the SAT 2015 Competition*, 2015.

[9] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT competition 2017. In *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, pages 14–15, 2017.

[10] Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 405–422. Springer, 2015.

[11] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[12] Vašek Chvátal and Bruce Reed. Mick gets some (the odds are on his side). In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, pages 620–627. IEEE, 1992.

[13] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM (JACM)*, 35(4):759–768, 1988.

[14] Amin Coja-Oghlan and Konstantinos Panagiotou. The asymptotic k-SAT threshold. *Advances in Mathematics*, 288:985–1068, 2016.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

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

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

[18] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing*, pages 61–75. Springer, 2005.

[19] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. Springer, 2003.

[20] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 251–262. ACM, 1999.

[21] Ehud Friedgut and Jean Bourgain. Sharp thresholds of graph properties, and the k-sat problem. *Journal of the American Mathematical Society*, 12(4):1017–1054, 1999.

[22] William I. Gasarch. Guest column: The P=?NP poll. *SIGACT News*, 33(2):34–47, 2002.

[23] Ian P. Gent and Peter Nightingale. A new encoding of AllDifferent into SAT. In *3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 95–110, 2004.

[24] Roman Gershman and Ofer Strichman. HaifaSat: A new robust SAT solver. In *Hardware and Software Verification and Testing, First International Haifa Verification Conference, Haifa, Israel, November 13-16, 2005, Revised Selected Papers*, pages 76–89. Springer, 2005.

[25] Jesús Giráldez-Cru and Jordi Levy. Generating SAT instances with community structure. *Artificial Intelligence*, 238:119–134, 2016.

[26] Eugene Goldberg and Yakov Novikov. BerkMin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[27] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In *Principles and Practice of Constraint Programming – CP97*, pages 121–135. Springer, 1997.

[28] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*, pages 431–437, 1998.

[29] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, pages 50–65. Springer, 2012.

[30] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[31] Antti E. J. Hyvärinen. *Grid based propositional satisfiability solving*. PhD thesis, Aalto University, 2011.

[32] David S. Johnson. Cliques, coloring, and satisfiability: second DIMACS implementation challenge. *DIMACS series in discrete mathematics and theoretical computer science*, 26:11–13, 1993.

[33] Hadi Katebi, Karem A. Sakallah, and João P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Theory and Applications of Satisfiability Testing – SAT 2011*, pages 343–356. Springer, 2011.

[34] Scott Kirkpatrick and Bart Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 1994.

[35] Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, and Yannis C. Stamatiou. Approximating the unsatisfiability threshold of random formulas. *Random Structures & Algorithms*, 12(3):253–269, 1998.

[36] Jia Hui Liang. *Machine Learning for SAT Solvers*. PhD thesis, University of Waterloo, 2018.

[37] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers. In *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, pages 225–241. Springer, 2015.

[38] Jia Hui Liang, Chanseok Oh, Minu Mathew, Ciza Thomas, Chunxiao Li, and Vijay Ganesh. Machine learning-based restart policy for CDCL SAT solvers. In *Theory and Applications of Satisfiability Testing – SAT 2018*, pages 94–110. Springer, 2018.

[39] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[40] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE, 1996.

[41] João P. Marques-Silva, Inês Lynce, and Sharad Malik. CDCL solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

[42] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 459–465, 1992.

[43] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[44] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on SAT solver performance. In *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 252–268. Springer, 2014.

[45] Steven David Prestwich. CNF encodings. In *Handbook of Satisfiability*, pages 75–97. IOS Press, 2009.

[46] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12(1): 23–41, 1965.

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

[48] Niklas Sörensson and Niklas Eén. Minisat v1.13 – a SAT solver with conflict-clause minimization. 2005.

[49] Grigori Tseitin. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.