CHAPTER TWENTY FIVE

On the Unreasonable Effectiveness of SAT Solvers

Vijay Ganesh and Moshe Y. Vardi

Abstract: Boolean satisfiability (SAT) is arguably the quintessential *NP*-complete problem, believed to be intractable in general. Yet, over the last two decades, engineers have designed and implemented conflict-driven clause learning (CDCL) SAT solving algorithms that can efficiently solve real-world instances with tens of millions of variables and clauses in them. Despite their dramatic impact, SAT solvers remain poorly understood. There are significant gaps in our theoretical understanding of why these solvers work as well as they do. This question of "why CDCL SAT solvers are efficient for many classes of large real-world instances while at the same time perform poorly on relatively small randomly generated or cryptographic instances" has stumped theorists and practitioners alike for more than two decades. In this chapter, we survey the current state of our theoretical understanding of this question, future directions, as well as open problems.

25.1 Introduction: The Boolean SAT Problem and Solvers

Boolean satisfiability (SAT) is one of the central problems in computer science and mathematics, believed to be intractable in general. This problem has been studied intensively by theorists since it was shown to be *NP*-complete by Cook (1971). The problem can be stated as follows:

Problem Statement 25.1 (The Boolean Satisfiability Problem) *Given a Boolean formula $\phi(x_1, x_2, \ldots, x_n)$ in conjunctive normal form (CNF) over Boolean variables $x_1, x_2, \ldots, x_n$, determine whether it is satisfiable. We say that a formula $\phi(x_1, x_2, \ldots, x_n)$ is satisfiable if there exists an assignment to the variables of $\phi(x_1, x_2, \ldots, x_n)$ such that the formula evaluates to true under that assignment. Otherwise, we say the formula is unsatisfiable. This problem is also sometimes referred to as CNF-SAT.*

There are many variations of the SAT problem that are all equivalent from a worst-case complexity-theoretic perspective. By SAT, we always refer to the CNF-SAT problem, unless otherwise stated. A SAT solver is a computer program aimed at solving the Boolean satisfiability problem.

More recently, practitioners who work in software engineering (broadly construed to include software testing, verification, program analysis, synthesis), computer security, and artificial intelligence (AI) have shown considerable interest in SAT solvers. This is due to the efficiency, utility, and impact of solvers on software engineering (Cadar et al., 2006), verification (Clarke et al., 2001), and AI planning (Kautz et al., 1992). The success of SAT solvers can be attributed to the fact that engineers have designed and implemented highly scalable conflict-driven clause learning (CDCL) SAT solving algorithms (or simply, SAT solvers[1]) that are able to efficiently solve multimillion variable instances obtained from real-world applications. What is even more surprising is that these solvers often outperform special-purpose algorithms designed specifically for the aforementioned applications. Having said that, it is also known that SAT solvers perform poorly on relatively small randomly generated or cryptographic instances (Balyo et al., 2017). Therefore, the key question in SAT solver research is:
"Why are CDCL SAT solvers efficient for many classes of real-world instances while at the same time perform poorly on randomly generated or cryptographic instances?"

This question has stumped theorists and practitioners alike for more than two decades. In order to address this question we have to go beyond traditional worst-case complexity and develop a parameterized complexity-theoretic understanding of real-world formulas over which CDCL solvers perform well. In this chapter, we survey the state of our knowledge vis-à-vis complexity-theoretic understanding of the power of SAT solvers as well as empirical studies of industrial instances that shed light on this central question.

25.1.1 The Central Questions

Here we list a set of key questions that are essential for a deeper understanding of SAT solvers, followed by sections where we discuss answers to them.

1. **Modeling SAT Solvers as Proof Systems** Perhaps the most important question in this context of understanding SAT solvers is the following: "What is an appropriate mathematical model for CDCL SAT solvers that explains both their efficacy as well as limitations?" This question is of paramount importance not only from a theoretical point of view, but as we argue in the text that follows, critical from a practical solver-design perspective as well. (We discuss detailed answers to this question in Section 25.3.)

   Over the past two decades, a consensus has developed among most theorists and practitioners that SAT solvers are best modeled as proof systems, i.e., a collection of proof rules and axioms. The history of this consensus is quite interesting and goes back to folklore theorems about Davis–Putnam–Logemann–Loveland (DPLL) SAT solvers (Davis et al., 1962) that state that they are essentially equivalent to tree-like resolution (for unsatisfiable inputs). Given that DPLL SAT solvers form the basis for the more powerful CDCL methods, the known connection between DPLL and tree-like resolution naturally led to the conjecture, and

[1] While researchers have studied a variety of algorithms for the Boolean SAT problem, in this chapter we focus only on sequential CDCL SAT solvers. The reason is that to date only these solvers seem to scale well for large real-world industrial instances, which is the key mystery addressed here.

subsequent proof, that CDCL (with nondeterministic variable/value selection and restarts) solvers are polynomially equivalent to general resolution, a proof system known to be stronger than tree-like resolution (Atserias et al., 2011; Pipatsrisawat and Darwiche, 2011). This result highlights the mathematical value of modeling solvers as proof systems. First, the "solvers-as-proof-systems" abstraction enables one to leverage powerful methods and results from proof complexity to obtain lower and upper bounds on the length of proofs constructed by solvers. Second, proof complexity suggests many different kinds of proof rules, e.g., extended resolution (Krajíček, 2019), that can be incorporated into solvers, thus strengthening them further. Finally, and perhaps most importantly, proof complexity enables a much deeper understanding of the power of certain solver heuristics (e.g., clause learning), that are best understood in terms of proof rules (general resolution).

While proof systems are a natural and elegant way to model SAT solvers, given that they are designed to construct proofs for unsatisfiable formulas, it is quite legitimate to ask whether they are suitable for studying solver behavior over satisfiable instances as well. It turns out that even when an input formula is satisfiable, SAT solvers generate proofs that establish unsatisfiability of the parts of the search space that do not contain satisfying assignments, thus guiding the solver away from fruitless parts of the search space (the set of all assignments). Hence, one could argue that proof systems are an excellent mathematical model for studying the complexity of SAT solvers for both satisfiable and unsatisfiable instances.

2. **Proof Search and SAT Solvers** While modeling solvers as proof systems enables us to prove upper and lower bounds, it does not quite address the issue of proof search. Proof systems, by their very nature, are best defined as nondeterministic objects. SAT solvers, however, are efficient implementations of proof systems. Hence, in order to properly frame the notion of proof search, we need to view solvers as optimization procedures that attempt to find the optimal (e.g., shortest) proof for a given input. Theorists refer to this as *automatizability of proof systems* (Bonet et al., 2000). Informally, we say that a proof system is automatizable if there exists an algorithm that can find a proof for a unsatisfiable formula with only polynomial overhead (in the size of the formula) over the optimal proof. (We address this in Section 25.4.)
3. **Parameteric Understanding of Boolean Formulas** The topic that has probably received the most attention in the context of understanding solvers is that of parameterization of Boolean formulas (i.e., parameters that enable us to classify formulas as easy or hard for solvers). This question can be restated as "What is a precise mathematical parametric characterization of real-world industrial or application instances over which SAT solvers perform well, and dually, of families of instances over which they perform poorly?" Examples of parameters that have been extensively studied include the clause-variable ratio, backdoors, backbones, community structure, and merge. We discuss the strengths and weaknesses of these parameters. The crucial requirement that these parameters have to satisfy is that they should be amenable to both theoretical analysis (i.e., enable parameterized complexity-theoretic bounds) as well as be relevant in practice (i.e., empirically explain the power of solvers and enable the design of better ones). We address this in Section 25.5. (See also Chapter 2 for a general discussion on parameterized complexity.)

4. **Proof Complexity and Solver Design** In addition to the aforementioned benefits of the "solvers-as-proof-systems" model, proof complexity theory also enables us to systematize practical solver design. Without the aid of proof complexity, practical solvers can seem like an incredibly complicated jumble of heuristics. When viewed, however, through the lens of proof systems, we can discern that certain solver heuristics correspond to proof rules (e.g., clause learning corresponds to the resolution proof rule), while other methods are aimed at optimally sequencing/selecting proof rules (e.g., branching) or (re)-initializing proof search (e.g., restarts). Further, solver heuristics that are aimed at sequencing proof rules or initializing proof search, can be profitably implemented using online, dynamic, and adaptive machine learning (ML) methods. In a series of papers, Liang et al. (2016, 2018) make exactly this point by designing and implementing a series of efficient ML-based CDCL SAT solvers. In Section 25.6, we discuss how theoretical concepts (proof systems) and practical insights (ML-based proof-rule sequencing/selection) can be brought together for better solver design. (See also Chapter 30 for ML-based algorithm design.)

25.2 Conflict-Driven Clause Learning SAT Solvers

In this section, we briefly describe the CDCL SAT solver (Marques-Silva and Sakallah, 1996; Moskewicz et al., 2001), whose pseudo code is presented in Algorithm 1. The CDCL algorithm is built on top of the well known DPLL method developed originally by Davis et al. (1962), and differs from it primarily in its use of the following heuristics: conflict analysis and clause learning (Marques-Silva and Sakallah, 1996), effective variable- and value-election heuristics (Moskewicz et al., 2001; Liang et al., 2016), restarts (Liang et al., 2018), clause deletion (Audemard and Simon, 2013), and lazy data structures (Moskewicz et al., 2001). The CDCL algorithm is a sound, complete, and terminating backtracking decision procedure for Boolean logic. It takes as input a Boolean formula ϕ in CNF and an initially empty assignment μ (aka assignment trail), and outputs SAT if the input formula ϕ has a solution, and outputs UNSAT otherwise.

Given the intricacies of the CDCL algorithm, it is difficult to describe its implementation in great detail in a few short pages. Instead, we focus on a conceptual and theoretically interesting presentation. For example, we discuss subroutines such as clause learning and Boolean constraint propagation (BCP) that are essential for a theoretical explanation of why solvers are efficient, rather than implementation of lazy data structures. Further, all our theoretical models have no clause deletion policy, partly because we have virtually no theoretical understanding of the impact of such policies on solver behavior.

Another important modelling decision often made is to assume that certain solver heuristics (e.g., restarts and variable/value selection) are nondeterministic or all-powerful. That is, for an unsatisfiable input, the dynamic choices made by these heuristics enable the CDCL solver to find the shortest proof (in the number of proof steps) of its unsatisfiability, with only a polynomial time overhead in proof search over the optimal for that input. Such modeling choices are very valuable for two reasons: First, they enable us to establish the strongest possible lower bounds (under nondeterministic assumptions), and second, they simplify the theoretical analysis.

Algorithm 1 The CDCL SAT solving algorithm

```
1:  function CDCL(φ, μ)
2:    Input: A CNF formula φ, and an initially empty assignment trail μ
3:    Output: SAT or UNSAT
4:
5:    dl = 0;                                          ▷ : Initially, decision level dl is 0
6:    if (CONFLICT == Boolean_Constraint_Propagation(φ,μ)) then
7:      return UNSAT;
8:    else if (all variables have been assigned) then
9:      return SAT;
10:   end if
11:   do                                                          ▷ The search loop
12:     x = DecisionHeuristic(φ,μ);          ▷ Variable- and value-selection heuristic
13:     dl = dl + 1;                                              ▷ : Increment dl
14:     μ = μ ⋃(x, dl);                          ▷ Add literal x to the assignment trail μ
15:     if (CONFLICT == Boolean_Constraint_Propagation(φ, μ)) then
16:       {β, C} = ConflictAnalysis(φ, μ);
17:                          ▷ Analyze conflict, learn clause C and backjump level β
18:       AddLearnedClause(C)
19:       if β < 0 then                                      ▷ β is the backjump level
20:         return UNSAT;                                      ▷ Top-level conflict
21:       else if (restart condition met) then
22:         restart;              ▷ dl is set to 0, and assignment trail μ is emptied
23:       else
24:         backtrack(φ, μ, β);                  ▷ Backjump to start search again
25:         dl = β;
26:       end if
27:     end if
28:   while (all variables have NOT been assigned)
29:   return SAT;
30: end function
```

Boolean Constraint Propagation (BCP) The CDCL algorithm first calls the BCP subroutine on input formulas without having branched on variables in it (line 6 in Algorithm 1). If a conflict is detected at this level (i.e., a top-level conflict), then CDCL returns UNSAT. The BCP subroutine (also referred to as unit propagation) is an incomplete SAT solver that takes as input a Boolean formula in CNF, and outputs SAT, CONFLICT, or UNKNOWN. It repeatedly applies the unit resolution rule to the input formula until it reaches a fixpoint. The unit resolution rule is a special case of the general resolution rule, where at least one of the clauses input to the rule is unit (i.e., contains exactly one unassigned literal under the *current partial assignment*). For example, consider the clauses (x) and $(\neg x \vee \alpha)$, which when resolved results in derived clause (α) written as: $(x) \quad (\neg x \vee \alpha) \vdash (\alpha)$. (We choose to use the symbol $\vdash$ to denote a derivation or proof step, with antecedents on its left side and consequent on its right side.)

Repeated applications of the unit resolution rule to an input formula, until reaching a fixpoint, amount to maintaining a queue of unit clauses, simplifying the

formula (along with all the learned clauses in the solver's learned clause database) with respect to the "current" unit clause (i.e., all occurrences of the current unit literal in the formula are assigned true, the occurrences of the complement of this unit literal are assigned false, and the clauses in the input formula are appropriately simplified), popping the "current" unit literal from the queue and adding implied units to the unit clause queue, and repeating this process until this queue is empty. A variable x that is assigned a value (alternatively, a variable whose value is set) as a result of applying BCP (one or more application of the unit resolution rule) is said to be *implied*/*propagated*.

BCP may return CONFLICT (i.e., the current partial assignment is unsatisfying for the input formula) or SAT (i.e., all variables have been assigned values true or false) or UNKNOWN. If BCP returns CONFLICT at the top level, without having made decisions (lines 7 and 20), then this means that the input formula is UNSAT. If, on the other hand, all the variables of the input formula have been assigned, then this means that the solver has found a satisfying assignment and it returns SAT (lines 9 and 29). Else, it means that the BCP subroutine returns UNKNOWN, i.e., it cannot decide by itself whether the formula is SAT or UNSAT. This causes the variable and value-selection heuristics to be invoked, that select an unassigned variable and assign it a truth value (line 12),[2] and iteratively search for a satisfying assignment to the input formula by extending the current partial assignment (line 11).

Variable- and Value-Selection Heuristics Variable selection heuristics[3] are subroutines that take as input the partial state of the solver (e.g., learned clauses and current partial assignment), compute a partial order over the unassigned variables of the input formula, and output the highest ranked variable in this order (line 12). Value-selection heuristics are subroutines that take as input a variable and output a truth value. After a variable selection heuristic selects an unassigned variable to branch on, the selected variable is then assigned a truth value given by a value-selection heuristic and added to the current partial assignment (line 14). Solver researchers have understood for a long time that both variable- and value-selection heuristics play a crucial role in the performance of solvers and a considerable amount of research has gone into their design (Liang et al., 2016, 2018). Unfortunately, due to space limitations we will only present a very brief sketch of the work done on this topic.

Decision Levels, Assignment Trail, and Antecedents On line 5 of the CDCL Algorithm 1, the solver initializes the variable dl (abbrev. for *current decision level*) to 0. The variable dl keeps track of the number of decisions in the *current partial assignment* as the solver traverses paths in the search tree of the input formula. Whenever a variable in the input formula is branched on (a decision variable), the variable dl in the CDCL algorithm is incremented by 1 (line 13). When the solver

[2] The variable being assigned by the solver's variable selection heuristic is sometimes also referred to as a branching or decision variable.

[3] Variable selection heuristics are sometimes also referred to as branching, with the variable output by them referred to as branching or decision variables. The term *decision heuristic* typically refers to the combination of variable- and value-selection heuristics. The literal returned by a decision heuristic is referred to simply as a *decision* or *decision literal*. The term *decision variable* refers to the variable that corresponds to a decision.

backjumps after ConflictAnalysis, the current decision level is modified to the level that the solver backjumps to (line 25).

The assignment trail (aka decision stack or partial assignment) μ is a stack data structure, where every entry corresponds to a variable, its value assignment, and its decision level. Whenever a variable x is branched or decided upon, an entry corresponding to x is pushed onto the assignment trail. Further, whenever the solver backjumps from level d to some level $d - \beta$, all entries with decision level higher than $d - \beta$ are popped from the assignment trail. The decision level of a variable is computed as follows: for unassigned variables it is initialized to -1. Unit variables in the input formula are assigned decision level 0. Whenever a variable is decided or branched upon, its decision level is set to dl + 1. Finally, the decision level of an implied literal x_i is the same as the decision level of the current decision variable in the assignment trail.

In addition to the decision level and the truth value assigned to a variable, the solver maintains another *dynamic object* for every variable x, namely, its *antecedent*. As the solver branches, propagates, backjumps, or restarts, the values this object takes may change. The antecedent or the *reason clause* of a variable x is the unit clause c (under the current partial assignment) used by BCP to *imply* x. For variables that are decisions or unassigned, the antecedent is NIL.

The Search Loop in CDCL If there is no conflict at the top level, i.e., dl=0 (line 6), then the algorithm checks whether all the variables of the input formula have been assigned a value (line 8). If so, the solver simply returns SAT. Else, it enters the body of the do-while loop on line 11, decides on a variable of the input formula using its variable- and value-selection heuristics (the DecisionHeuristic subroutine on line 12), increments the decision level (line 13), pushes the decision variable to the partial assignment or the assignment trail μ, along with its decision level (line 14), and performs BCP (line 15). If BCP returns CONFLICT (i.e., the current assignment μ is unsatisfying for the input formula), then conflict analysis is triggered (line 17).

Conflict Analysis and Clause Learning The conflict analysis and clause learning subroutine is perhaps the most important part of a CDCL SAT solver. The ConflictAnalysis subroutine (line 17) determines a reason or root cause for the conflict, learns a corresponding conflict or learned clause, and computes the *backjump level*. Most CDCL solvers implement what are known as *asserting clause learning schemes*, i.e., ones that learn clauses containing exactly one variable from the highest decision level (we discuss this in more detail in Section 25.3). If the backjump level is below 0, then the solver returns UNSAT (line 20), since this corresponds to deriving false. Otherwise, the CDCL solver may jump back several decision levels in the search tree (line 24), unlike in the DPLL case where the solver backtracks only one decision level.

One of the simplest forms of clause learning is the Decision Learning Scheme (DLS). While it is not the most effective (that honor goes to the first-UIP method by Moskewicz et al., 2001), DLS is certainly easy to explain. The key idea behind DLS can be explained as follows: All solvers, irrespective of the asserting learning scheme they implement, maintain a directed acyclic graph of implications whose nodes are variables (either decision or propagated), and there is an edge from node a to node b if setting a caused BCP to set b as well, under the current partial assignment.

Whenever the solver detects a conflict, this graph is analyzed by the ConflictAnalysis subroutine with the goal of determining the root cause of said conflict. In DLS, the ConflictAnalysis subroutine simply takes the negation of the conjunction of decisions responsible for the conflict, and the learned clause thus computed is stored in a learned clause database. Such learned clauses prevent subsequent invocations of BCP from making the same combination of mistakes (i.e., decisions) that led to the conflict. This process is repeated until the solver correctly determines the satisfiability of the input formula.

Backjumping In its simplest form, the backtracking step in a DPLL solver works as follows: on reaching a conflict, these solvers undo the last decision that led to the conflict, which leads the solver to backtrack to the previous decision level and continue its search. In the context of CDCL solvers many backtracking methods have been explored. Perhaps the most well known is called nonchronological backtracking (or simply, backjumping), wherein the solver backjumps to the second-highest decision level over all the variables in the asserting learned clause. Jumping to the second-highest decision level has the benefit that the asserting clause is now unit under the "current" partial assignment (post backjump).[4]

Restarts The original restart heuristic was first proposed by Gomes et al. (1998), in the context of DPLL SAT solvers. The idea behind restart policies is very simple: The assignment trail of a solver is erased at carefully chosen intervals during its run (with the exception of learned unit clauses that are not deleted). It is well known that restarts are a key solver heuristic, both empirically and theoretically. The original idea behind restarts, referred to as the heavy-tailed distribution explanation, is that SAT solvers have variance in their run time due to randomization, and may get unlucky resulting in uncharacteristically long run times. A restart in such cases gives the solver a second chance of getting a shorter run time. This explanation has now been partially discarded in favor of an empirically more robust argument that restarts enable solvers to learn better learned clauses, since they shorten the assignment trail at several intervals during the solver's run (Liang et al., 2018). On the theoretical front, recent work by Bonet et al. (2014) showed that CDCL SAT solvers with no restart (but with nondeterministic variable and value selection) are strictly more powerful than regular resolution. Nevertheless, the question of why restarts are so important for efficiency of CDCL SAT solvers remains open, both from the theoretical and empirical perspectives.

25.3 Proof Complexity of SAT Solvers

25.3.1 Equivalence between CDCL and Resolution

In this subsection, we survey known results regarding the "SAT solver as a proof system" model. Specifically, we discuss the seminal simulation result by Pipatsrisawat and Darwiche (2011) and independently by Atserias et al. (2011), who showed that

[4] While we do not discuss clause deletion policies at length, they do deserve mention, since they are an important heuristic in the context of CDCL SAT solvers. The key purpose of deletion policies is removal of derived or learned clauses that have outlived their utility vis-à-vis proof search. As is probably already clear to the reader, predicting the utility of derived clauses is a very difficult problem in general.

CDCL SAT solvers (with nondeterministic branching, restarts, and asserting learning scheme) are polynomially equivalent to the general resolution (Res) proof system. The history of these simulation results go back to the paper by Beame et al. (2004), who first showed that CDCL SAT solvers (under the assumption that a solver can branch on a variable that is already assigned a truth value) are polynomially equivalent to general resolution.

While theorists had long anticipated a polynomial equivalence between CDCL SAT solvers and the general resolution proof system, it was not formally established until 2011 (Atserias et al., 2011; Pipatsrisawat and Darwiche, 2011). In their seminal work, Pipatsrisawat and Darwiche, as well as Atserias et al. realized that CDCL solvers simulate Res not necessarily by producing the Res-proofs exactly, but rather by "absorbing" the clauses of Res-proofs. One should think of the absorbed clauses as being "learned implicitly" – absorbed clauses may not necessarily appear in a formula $\mathcal{F}$ or its proof. If we assign, however, all but one of the literals in the clause to false then unit propagation in CDCL will set the final literal to true. That is, even if the absorbed clause C is not in $\mathcal{F}$, the unit propagation subroutine behaves "as though" the absorbed clause is actually in $\mathcal{F}$. The dual of the notion of absorbed clauses is the concept of 1-empowering clauses.[5] Informally, 1-empowering clauses are ones that have not been "learned implicitly," and may enable BCP to make progress. We now define these notions more precisely, followed by a sketch of the main idea behind the simulation proof.

Definition 25.2 (Asserting Clause) Recall that an *assignment trail* is a sequence of pairs $\sigma = \{(\ell_1, d_1), (\ell_2, d_2), \ldots, (\ell_t, d_t)\}$ where each literal ℓ_i is a literal from the formula and each $d_i \in \{\mathsf{d}, \mathsf{p}\}$, indicating that the literal was set by the solver by a decision or by a unit propagation, respectively. The *decision level* of a literal ℓ_i in the branching sequence is the number of decision literals occurring in σ up to and including ℓ_i. The *state* of a CDCL solver at a given point during its run can be defined as $(\mathcal{F}, \Gamma, \sigma)$, where $\mathcal{F}$ is the input CNF formula, Γ is a set of learned clauses, and σ is the assignment trail at the given point during the solver's run. Given an assignment trail σ and a clause C we say that C is *asserting* if it contains exactly one literal occurring at the highest decision level in σ. A clause learning scheme is *asserting* if all conflict clauses produced by the scheme are asserting with respect to the assignment trail at the time of conflict.

Definition 25.3 (Extended Branching Sequence) An *extended branching sequence* is an ordered sequence $B = \{\beta_1, \beta_2, \ldots, \beta_t\}$ where each β_i is either a branching literal, or a symbol R, denoting a restart. If A is a CDCL solver, we use an extended branching sequence to dictate the operation of the solver A on $\mathcal{F}$: whenever the solver calls the branching scheme, we consume the next β_i from the sequence. If it is a literal, then we branch on that literal appropriately; otherwise restart as dictated by the extended branching sequence. If the branching sequence is empty, then simply proceed using the heuristics defined by the algorithm.

[5] The idea of 1-empowering clauses was first introduced by Pipatsrisawat and Darwiche (2011), while its dual notion of absorbed clauses was introduced by Atserias et al. (2011).

Definition 25.4 (Unit Consistency) We say CNF formula $\mathcal{F}$ is unit inconsistent if and only if there is a proof of the unsatisfiability of $\mathcal{F}$ using only unit resolution (alternatively, via BCP). A formula that is not unit inconsistent is said to be unit consistent (sometimes also written as 1-consistent).

Definition 25.5 (1-Empowering Clauses) Let $\mathcal{F}$ be a set of clauses and let A be a CDCL solver. Let $C = (\alpha \Rightarrow \ell)$ be a clause, where α is a conjunction of literals. We say that C is *empowering with respect to* $\mathcal{F}$ *at* ℓ if the following holds: (1) $\mathcal{F} \models C$, (2) $\mathcal{F} \wedge \alpha$ is unit consistent, and (3) an execution of A on $\mathcal{F}$ that falsifies all literals in α does not derive ℓ via unit propagation (aka, BCP). The literal ℓ is said to be *empowering*. If item (1) is satisfied but one of (2) or (3) is false then we say that the solver A and $\mathcal{F}$ *absorbs* C at ℓ; if A and $\mathcal{F}$ absorbs C at every literal then the clause is simply *absorbed*.

Definition 25.6 (General and Tree-like Resolution Proofs) A general resolution proof can be defined as a directed acyclic graph (DAG), whose nodes are clauses which are either input or derived, and there is an edge from nodes A and B to C if C is derived from A, B via the resolution proof rule. Let $(\alpha \vee x)$ and $(\neg x \vee \beta)$ denote two clauses, where α, β are disjunction of literals. Then, the resolution proof rule derives $(\alpha \vee \beta)$, and is usually written as

$$(\alpha \vee x) \quad (\neg x \vee \beta) \vdash (\alpha \vee \beta).$$

We assume α, β do not contain opposing literals. A tree-like resolution proof is a restricted form of general resolution proof where the proofs may not share sub-proofs, i.e., they are tree-like.

In order for a clause C to be learned by a CDCL solver, it must be *1-empowering at some literal* ℓ at the point in time it is learned by the solver during its run. To see this, consider a trace of a CDCL solver, stopped right after it has learned a clause C. Since we have learned C it is easy to see that it must be the case that $\mathcal{F} \models C$. Let σ be the branching sequence leading to the conflict in which we learned C, and let ℓ be the last decision literal assigned in σ *before* the solver hit a conflict (if CDCL uses an asserting clause learning scheme, such a literal must exist). We can write $C \equiv (\alpha \Rightarrow \neg\ell)$, and clearly $\alpha \subseteq \sigma$. Thus, at the point in the branching sequence σ before we assign ℓ it must be that $\mathcal{F} \wedge \alpha$ is unit consistent, since we have assigned another literal after assigning each of the literals in α. Finally, $\mathcal{F} \wedge \alpha \nvdash_1 \ell$, since $\neg\ell$ was chosen as a decision literal *after* we set the literals in α. (By $\alpha \vdash_1 \beta$ we mean that the literal β is derived from the set of clauses α using only BCP.)

Definition 25.7 (1-Provable Clauses) Given a CNF formula $\mathcal{F}$, clause C is 1-provable with respect to $\mathcal{F}$ iff $\mathcal{F} \wedge \neg C \vdash_1$ *false*. Put differently, we say a clause C is 1-provable with respect to a CNF $\mathcal{F}$, if C is derivable from $\mathcal{F}$ only using BCP.

Theorem 25.8 *CDCL is polynomially equivalent to general resolution (Res).*

Proof Sketch The high level idea of the simulation is as follows: We need to show that for a Res proof of unsatisfiability of an input formula $\mathcal{F}$, the CDCL

solver (with nondeterministic extended branching sequence and asserting clause learning scheme) can simulate that proof with only polynomial overhead in the proof size (in terms of number of clauses). The crucial insight here is that for formulas $\mathcal{F}$ for which BCP alone cannot establish unsatisfiability, there exist empowering clauses implied by $\mathcal{F}$ that, when added to it (i.e., to the solver's database of learned clauses), cause BCP to correctly determine that the input formula is UNSAT. Further, for a general resolution proof π of a 1-consistent formula $\mathcal{F}$, there exists a clause C in π that is both 1-empowering and 1-provable with respect to the formula (at the point in the proof π that C is derived). Finally, such a clause can be absorbed by a CDCL solver in time $O(n^4)$, where n is the number of variables in the input formula. This process is repeated until there are no more clauses that need to be absorbed, and thus we have that CDCL polynomially simulates general resolution. (The reverse direction is trivial.) □

Discussion The value of Theorem 25.8 is threefold: First, we can easily lift lower bounds from the proof complexity literature for Res to CDCL SAT solvers, thus addressing the question of why solvers fail. Further, the polynomial equivalence between CDCL and Res helps explain the power of clause learning, since clause learning in CDCL corresponds to applications of the general resolution rule. In other words, proof complexity enables an improved understanding of certain heuristics in SAT solvers. Finally, proof complexity theory is a storehouse of knowledge on proof systems that can be leveraged to build solvers with varying degrees of strength aimed at different kinds of applications such as cryptography or verification.

25.3.2 Lower and Upper Bounds for Res and CDCL SAT Solvers

Considerable work has been done on the proof complexity of Res. Unfortunately, we cannot possibly do justice to this subject in this chapter. We do, however, sketch a few results that are relevant in the context of CDCL SAT solvers. The first superpolynomial lower bound for resolution was proved by Haken (1985). To be more precise, Haken proved that the family of formulas that encodes the *Propositional Pigeonhole Principle (PHP)* requires resolution proofs of size at least c^n, for some $c > 1$. Another source of hardness for Res comes from randomly generated formulas. Also, Urquhart showed that CNF formulas whose graphs are expanders are hard for Res, and hence for CDCL (Urquhart, 1987).

There is a vast literature on the complexity of proof systems that we have not covered here (Krajíček, 2019). For example, there are many powerful proof systems such as extended resolution with no known lower bounds, that have been studied extensively by theorists (Tseitin, 1983). While there are systems that are stronger than Res, their implementations to date as solvers seem to be useful only in narrow contexts, unlike CDCL SAT solvers that is widely applicable. This suggests that strength of the proof system alone may not lead to powerful solvers. One also has to look into proof search, which we turn to in the next section.

25.4 Proof Search, Automatizability, and CDCL SAT Solvers

Proof complexity gives us powerful tools that enable us to prove lower bounds for SAT solvers (and thus characterize families of formulas where solvers fail

spectacularly). It does not, however, quite address the question of proof search. The proof search question for a proof system is: Given an unsatisfiable formula F, does there exist an algorithm that finds proofs in the given proof system with only polynomial overhead? In particular, if the formula F has a short proof, then the question asks whether a solver can find a proof in polynomial time.

This idea of efficient proof search was first formalized by Bonet et al. (2000) via the notion of automatizability of proof systems. (Although there is previous work on proof search by Iwama (1997) in which it was shown that the problem of finding the shortest Res proof is *NP*-hard.) Recall that the polynomial simulation result in Section 25.3 shows that if there is a short proof π for some formula φ, then there exists a run of a nondeterministic CDCL solver (i.e., a CDCL SAT solver with nondeterministic variable/value selection and restarts) that can produce a proof of size $O(n^4) * |\pi|$. The proof of the theorem relies on the fact that the CDCL solver under consideration has nondeterministic power, yet real-life solvers do not have the same luxury. So, it is natural to ask the following question: "For the class of formulas that do have short proofs, does there exist a solver that always finds such proofs in time polynomial in the size of the input formula?"

In their seminal paper, Bonet et al. (2000) defined the notion of the *automatizability* of proof systems. A proof system P is said to be *automatizable* if there exists a polynomially bounded deterministic algorithm A that takes as input an unsatisfiable formula φ, and outputs a P-proof of φ of size at most polynomially larger (in the size of φ) than the shortest P-proof of φ. There have been several attempts to tackle the automatizability problem for resolution and tree-like resolution. For example, Ben-Sasson and Wigderson (2001) showed that tree-like resolution is automatizable in quasi-polynomial time. A more recent breakthrough result by Atserias and Müller (2019) says that Res is not automatizable in quasi-polynomial time, unless *NP* is included in SUBEXP.

Automatizability and CDCL SAT Solvers The value of studying the (parametric) automatizability question is that it may eventually shed light on the key question of upper bounds (i.e., why are solvers efficient for certain classes of industrial instances), just as proof complexity of the Res system has helped us better understand lower bounds for CDCL solvers. Automatizability gets to the heart of the proof search question, precisely the task that solvers have been designed for. While we are far from any conclusive answers, we do have promising leads. For example, based on the quasi-automatizability result for tree-like resolution (equivalently, DPLL solvers), we know that if an unsatisfiable formula has a polynomial-sized tree-like proof, DPLL solvers can solve it in quasi-polynomial time. One could ask whether something similar is also (parameterically) true for general resolution and equivalently for CDCL solvers. This naturally leads us to a parameteric study of formulas and their proofs.

25.5 Parameteric Understanding of Boolean Formulas

So far we have addressed the issue of how best to model SAT solvers as proof systems, discussed lower bounds vis-à-vis proof size obtained via an equivalence between solvers and proof systems, and lower bounds for automatizability of Res. While these give us insight into classes of instances over which SAT solvers perform poorly, they

do not quite address the central question of solver research, namely, why CDCL SAT solvers are so efficient for instances obtained from real-world applications. In order to better understand this question we need to turn our attention to parameterization of Boolean formulas and their proofs.

There is widespread consensus among SAT researchers that solvers somehow leverage the structure present in real-world CNF formulas (or in their proofs), and that a characterization of this structure can be used to establish a parameteric proof-complexity theoretic and proof search upper bounds. As a consequence, considerable effort has been expended in studying the structure of real-world formulas. We already know that parameterizations such as 2-SAT or Horn clauses that are known to be easy for Res, do not really capture classes of real-world instances over which solvers perform well.

In fact, the challenge for this line of research is to come up with parameters that make sense both in practice (i.e., characterize the structure of real-world instances) and theory (i.e., are amenable to theoretical analysis). While researchers have proposed several parameters, none so far seems to be up to the task of addressing this challenge. The parameters that are easy to work with in theory (e.g., backdoors) do not seem to characterize real-world instances. The ones that do seem to characterize real-world instances (e.g., community structure or modularity) are difficult to work with from a theoretical point of view. Even so, there are many lessons one can learn from the parameters studied so far that may eventually help prove the kind of parameteric upper bounds on proof complexity and proof search that we seek.

Clause Variable Ratio Perhaps one of the first and certainly most widely studied parameter is the *Clause/Variable Ratio (CVR) or Clause Density*, given its intuitive appeal. The CVR of a k-CNF formula is defined as the ratio of the total number of clauses to the number of variables in the formula. The earliest experiments in regards to CVR were performed by Cheeseman et al. (1991), who showed that for randomly generated fixed-width CNF formulas the probability of satisfiability of such instances undergoes a phase-transition around a fixed CVR, which depends only on clause-width (the phase-transition for randomly generated 3-CNF formulas is 4.26). Formulas below the phase transition are more likely to be satisfiable (the asymptotic probability approaches 1 as the CVR goes below 3.52 (Kaporis et al., 2006)), and those above are more likely to be unsatisfiable (the asymptotic probability approaches 1 as the CVR goes above 4.4898 (Díaz et al., 2009)). Further, it was observed that formulas below and above the phase transition are easy to solve, while those around the phase transition are hard (the so-called "easy-hard-easy pattern") (Mitchell et al., 1992).

These results caused a stir when first reported, for it seemed like there was a very simple explanation for the worst-case complexity-theoretic hardness of the Boolean satisfiability problem. It was soon, however, observed that there are many issues with these results. First, it is known that phase transitions exists also for SAT problems that are known to be easy, such as 2-SAT; cf. Chvátal and Reed (1992). Second, when one takes a deeper look at the empirical "easy-hard-easy" pattern of difficulty of solving instances around the phase transition, by keeping the CVR constant and scaling the size of instances, the observed pattern is more like "easy-harder-less hard" (Coarfa et al., 2003), with the transition from "easy" to "harder" occurring well in

the satisfiable region. This empirical finding was later confirmed in Achlioptas and Coja-Oghlan (2008), who demonstrated a "shattering" of the solution space around CVR of 3.8.

Treewidth The *treewidth* of a graph measures how close a given graph is to being a tree (Bodlaender, 1994). A tree has treewidth 1. A cycle is the simplest graph that is not a tree, but it can be "squashed" into a path of treewidth 2. A family of graphs is of *bounded treewidth* if there some some $k > 0$ such that all graphs in the family has treewidth at most k. It turns out that many graph problems that are *NP*-hard on general graphs can be solved in polynomial time on bounded-treewidth families of graphs (Freuder, 1990). This idea can also be applied to SAT. Given a CNF forumla ϕ, we can construct a bipartite graph G_ϕ whose nodes are the clauses and the variables of ϕ, and there is an edge between a clause c and a variable v when v occurs in c. The treewidth of ϕ is then the treewidth of G_ϕ. It follows that that SAT can be solved in polynomial time for a bounded-treewidth families of formulas.

If industrial formulas had bounded treewidth, that would perhaps explain the success of CDCL solvers on such formulas. For example, formulas generated by *bounded model checkers* are obtained by unrolling a circuit (Clarke et al., 2001), which yields formulas that have bounded treewidth (in fact, even bounded pathwidth) (Ferrara et al., 2005). It is not clear, however, that this explanation is satisfactory. Polynomial-time algorithms for graph families of treewidth at most k, typically have worst-case time complexity of the form $n^{O(k)}$ (Kolaitis and Vardi, 2000). Thus, such polynomial-time algorithm are feasible in practice only for very small k's, which does not seem to be the case, for example, in bounded model checking.

Backdoors and Backbones The notion of *backdoors* for Boolean formulas was first introduced by Williams et al. (2003). The intuition behind this notion is quite elegant, namely, that for every Boolean formula there is a (small) subset of its variables, which when assigned appropriate values, renders the said formula easy to solve. It was further conjectured that industrial instances must have small backdoors. Williams et al. introduced two kinds of backdoors, namely *weak backdoors* and *strong backdoors*. A weak backdoor B of a satisfiable formula φ is a subset of variables of φ, where there exists a mapping $\delta : B \mapsto \{0,1\}$, such that the restricted formula $\varphi[\delta]$ can be solved in polynomial time by some subsolver S (e.g., BCP). By contrast, a strong backdoor B of a formula φ is a subset of variables from φ, such that for a mapping $\delta : B \mapsto \{0,1\}$ from variables in B to truth values, the restricted formula $\varphi[\delta]$ can be solved by a polynomial time subsolver. While weak backdoors are defined only for satisfiable instances, strong backdoors are well defined for both satisfiable and unsatisfiable ones. The *backbone* of a satisfiable Boolean formula φ can be defined as a subset B of variables such that they take the same values in all satisfying assignments and the set B is maximal. Kilby et al. (2005) theoretically proved that backbones are hard to even approximate assuming $P \neq NP$. Unfortunately, both backdoors (and backbones) do not seem to explain why industrial instances are easy. Often industrial instances seem to have large backdoors (Zulkoski et al., 2018b). Further, the hypothesized correlation between the size of backdoors and solver runtime (i.e., smaller the backdoor, easier the problem) seems weak at best for industrial instances (Zulkoski et al., 2018b). It seems like CDCL SAT solvers are not able to automatically identify and exploit small backdoors or backbones.

Modularity and Community Structure Another structure that has been extensively studied is the community structure of the variable-incidence graph or VIG of CNF formulas (formula variables correspond to nodes in the VIG, and there is an edge between two nodes if the corresponding variables appear in the same clause). Informally, community structure of a graph defines how "separable" the clusters of a graph are. An ideal clustering for a VIG would be where every cluster corresponds to a set of clauses that are easy to solve independently and the clusters are "weakly connected" to other clusters. The concept of dividing graphs into natural communities was developed by Clauset et al. (2004). On a high level, we say a graph G has good community structure, that is, there is an optimal decomposition of G (we call each subgraph/component a community/module) such that there are far more intracommunity edges than there are intercommunity ones. Clauset et al. defined the notion of modularity of a graph, denoted Q, more specifically, a graph with high Q value is more "separable" (in the sense that the communities have few intercommunity edges relative to the number of communities) comparing to a graph with low Q value (which is closer to a randomly generated graph).

In their seminal paper, Ansótegui et al. (2012) established that industrial instances have good community structure. Newsham et al. (2014) showed a strong correlation between community structure and solver performance. Specifically, they showed that formulas with good community structure (high Q) correlate with lower solver run time relative to formulas with low Q. Subsequent work has taken advantage of these results in coming up with better solver heuristics. Nevertheless, the promise of community structure as a basis for a theoretical understanding of why solvers perform well has not yet been realized.

Merge Resolution and Mergeability Aside from the parameters we have discussed so far, "merge" is another interesting parameter that researchers have studied from both theoretical (Andrews, 1968) and practical (Zulkoski et al., 2018a) points of view. We motivate the study of merge parameter by recalling the resolution rule given above. Let A denote the antecedent clause $(\alpha \vee x)$, B denote the antecedent clause $(\neg x \vee \beta)$, and C denote the derived clause or consequent $(\alpha \vee \beta)$. For a clause A, let $\mid A \mid$ denote the number of literals in it (the length of the clause). It is easy to see that the length $\mid C \mid$ of the consequent C is equal to $|A|+|B|-l-2$, where $l = \mid A \cap B \mid$ is the number of literals overlapping in A and B. Put differently, the length of derived clauses in the Res proof system decreases as the number of literals overlapping in the antecedents increase. This number l of overlapping literals in the antecedent of a resolution proof rule application is called *merge*.

We can further make the following observations about the relationship between merge and the completeness of the Res proof system: First observe that derived clauses in a resolution proof decrease in length proportional to the increase in merge (i.e., the overlap between antecedent clauses). Additionally, in order for a Res proof of unsatisfiability to terminate, the length of derived clauses have to start "shrinking" (i.e., the derived clause is strictly smaller than at least one of its antecedent) at some point in the proof, eventually ending in the empty clause. It turns out that repeated application of the resolution rule over clauses with large merge is a powerful way to obtain short clauses, eventually enabling the resolution proof system to obtain complete proofs.

In fact, the power of merge was first observed by Andrews (1968), who defined merge resolution as a refinement of the Res proof system. The merge resolution proof system, which is sound and complete for propositional logic, is designed to bias applications of the resolution proof rule over clauses that have high degree of merge, and thus obtain shorter derived clauses faster relative to a proof system that is not biased in this way. Intuitively, this is a powerful greedy heuristic since maximizing merge likely implies that the *resolution width* of a proof (Ben-Sasson and Wigderson, 2001) also goes down during proof search. Nevertheless, a formal link between merge and resolution width remains to be established.

On the empirical front, Zulkoski et al. (2018a) studied the link between merge and efficiency of CDCL SAT solvers on randomly generated and industrial formulas. They defined a new notion called *mergeability* as follows: Let $m(A, B)$ be the number of overlapping literals in some resolvable pair of clauses A and B, and define M to be $\sum m(A, B)$ for all resolvable pairs of A and B. Additionally, let l be the number of clauses in the input formula ϕ. Then the mergeability of ϕ is defined as $\frac{M}{l^2}$. The empirical hypothesis they posed in their work was: "As the mergeability increases for a formula ϕ (while most of its other key features remains unmodified), the formula becomes easier to solve."

In their paper, Zulkoski et al. (2018a) report that indeed this is the case. They present a random industrial-like instance generator that takes as input a formula and then increases the mergeability of the formula while maintaining other key properties of the formula such as the distribution of variable occurrences, property of the underlying community structures, etc. It turns out, under their notion of mergeability, the runtime of CDCL SAT solvers negatively correlates with mergeability over randomly-generated unsatisfiable instances. Another observation they make is that the CDCL solvers they used in their experiments produce shorter and shorter width clauses on average as the mergeability of the input formula increases. These experiments strongly suggest that merge might be a key parameter that can help explain the power of CDCL solvers on industrial instances.

The jury is still out on how to prove a meaningful upper bound result that is relevant in practice and illuminating from a theoretical point of view. It is clear that we need parameterization. It is not, however, clear which of the aforementioned parameters will do the trick. Our conjecture is that the upper bound is likely to be exponential in both the parameter(s) and the size of the input (n, number of variables). Nevertheless, the parameter(s) and size n may interact in such a way that for relatively small values of n, the upper bound may behave like a polynomial, and for large values of n, the upper bound may behave more like an exponential.

25.6 Proof Complexity, Machine Learning, and Solver Design

For most of this chapter we have focused on a proof-theoretic model of CDCL SAT solvers and questions of lower/upper bounds of proof size and search. As we close, it behooves us to reflect on how these theoretical investigations may help us with practical solver design. If one were to investigate the source code of a typical CDCL SAT solver, without the aid of proof complexity, it is likely they will see a difficult-to-understand jumble of heuristics. Fortunately, a proof complexity-theoretic view can help appropriately abstract solver design.

While SAT solvers are decision procedures, internally they are an interacting set of complex optimization heuristics whose aim is to minimize solver runtime. Many solver heuristics correspond to proof rules (e.g., BCP corresponds to repeated application of the unit-resolution rule, while clause learning correspond to the general resolution rule), while others such as branching heuristics correspond to sequencing or selection of proof rules, and restarts correspond to initialization of proof search. This view suggests a solver-design principle: Solvers are best designed by understanding what kind of proof system best fits the application at hand, and developing optimization procedures to sequence, select, and initialize proof rules. These optimization procedures can be implemented using a rich set of known online and adaptive machine-learning methods. This empirical principle was properly articulated in a series of papers by Liang et al. (2016, 2018) that in turn led to the design and development of MapleSAT, one of the fastest solvers in recent years.

25.7 Conclusions and Future Directions

The question of why CDCL SAT solvers are efficient for industrial instances, while at the same time perform poorly on certain families of crafted and randomly generated instances, is one of the central questions in SAT-solver research (Vardi, 2014). We discussed how proof and parameterized complexity provide the appropriate lens through which we can hope to answer this question. The strongest result to date states that CDCL solvers (with nondeterministic branching and restarts) are as powerful as the Res proof systems. This simulation answers, to some extent, the question of why solvers fail on certain classes of instances by lifting known lower bounds for Res to the CDCL setting. Proof complexity also formalizes the question of proof search via the notion of automatizability, which can be a powerful lens through which to understand paramterized upper bounds on CDCL proof search. We also discussed the search for parameters that may be relevant both in practice and theory. The most promising among them are the merge and community structure parameters. Having said that, much progress needs to be made for we still do not know the right parameterization(s) for industrial instances. Finally, we discussed how solvers can be viewed as a collection of interacting heuristics, some of which implement appropriate proof rules, while others perform the task of proof rule sequencing, selection, and initialization, many of which can be profitably implemented via online and adaptive machine-learning techniques.

While much progress has been made, the central questions remain unanswered. We hope that this chapter suitably captures the progress made thus far, and frames the appropriate ideas that may lead to breakthrough results in the near future. Perhaps the most important unanswered question is that of appropriate parameterization(s) of industrial instances. Despite more than two decades of efforts by a number of leading practitioners and theorists, we still do not have good candidate parameters with which to upper bound the proof size and proof search for industrial instances. Another open problem that has resisted all attempts at solving is the question of the power of restarts (i.e., why are restarts so important in practice, and do they give solvers proof-theoretic power?). Finally, there are mysteries such as power of local branching (a la, the VSIDS heuristic) and first-UIP clause learning schemes. These heuristics seem indispensable and yet no one can convincingly explain why.

References

Achlioptas, Dimitris, and Coja-Oghlan, Amin. 2008. Algorithmic barriers from phase transitions. *2008 49th Annual IEEE Symposium on Foundations of Computer Science*, pp. 793–802. IEEE.

Andrews, Peter B. 1968. Resolution with merging. *Automation of Reasoning*, pp. 85–101. Springer.

Ansótegui, Carlos, Giráldez-Cru, Jesús, and Levy, Jordi. 2012. The community structure of SAT formulas. In Cimatti, Alessandro, and Sebastiani, Roberto (eds), *Theory and Applications of Satisfiability Testing – SAT 2012*, pp. 410–423. Springer.

Atserias, Albert, and Müller, Moritz. 2019. Automating resolution is NP-hard. In *60th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 498–509.

Atserias, Albert, Bonet, Maria Luisa, and Esteban, Juan Luis. 2002. Lower bounds for the weak pigeonhole principle and random formulas beyond resolution. *Information and Computation*, **176**(2), 136–152.

Atserias, Albert, Fichte, Johannes Klaus, and Thurley, Marc. 2011. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, **40**, 353–373.

Audemard, Gilles, and Simon, Laurent. 2013. Glucose 2.3 in the SAT 2013 competition. In *Proceedings of SAT Competition 2013*, pp. 42–43.

Balyo, Tomás, Heule, Marijn J. H., and Järvisalo, Matti. 2017. SAT competition 2016: Recent developments. In Singh, Satinder P., and Markovitch, Shaul (eds), *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 5061–5063. AAAI Press.

Beame, Paul, Kautz, Henry, and Sabharwal, Ashish. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, **22**, 319–351.

Ben-Sasson, Eli, and Wigderson, Avi. 2001. Short proofs are Resolution made simple. *Journal of the ACM*, **48**(2), 149–169.

Bodlaender, Hans L. 1994. A tourist guide through treewidth. *Acta Cybernetica*, **11**(1-2), 1.

Bonet, Maria Luisa, Pitassi, Toniann, and Raz, Ran. 2000. On interpolation and automatization for Frege systems. *SIAM Journal on Computing*, **29**(6), 1939–1967.

Bonet, Maria Luisa, Buss, Sam, and Johannsen, Jan. 2014. Improved separations of regular resolution from clause learning proof systems. *Journal of Artificial Intelligence Research*, **49**, 669–703.

Cadar, Cristian, Ganesh, Vijay, Pawlowski, Peter M., Dill, David L., and Engler, Dawson R. 2006. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322–335. CCS '06. ACM.

Cheeseman, Peter C, Kanefsky, Bob, and Taylor, William M. 1991. Where the really hard problems are. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 331–337.

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

Clarke, Edmund, Biere, Armin, Raimi, Richard, and Zhu, Yunshan. 2001. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, **19**(1), 7–34.

Clauset, Aaron, Newman, M. E. J., and Moore, Cristopher. 2004. Finding community structure in very large networks. *Physical Review E*, **70**(Dec), 066111.

Coarfa, Cristian, Demopoulos, Demetrios D., Aguirre, Alfonso San Miguel, Subramanian, Devika, and Vardi, Moshe Y. 2003. Random 3-SAT: The plot thickens. *Constraints*, **8**(3), 243–261.

Cook, Stephen A. 1971. The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pp. 151–158. ACM.

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

Díaz, Josep, Kirousis, Lefteris, Mitsche, Dieter, and Pérez-Giménez, Xavier. 2009. On the satisfiability threshold of formulas with three literals per clause. *Theoretical Computer Science*, **410**(30-32), 2920–2934.

Ferrara, Andrea, Pan, Guoqiang, and Vardi, Moshe Y. 2005. Treewidth in verification: Local vs. global. *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 489–503. Springer.

Freuder, Eugene C. 1990. Complexity of K-tree structured constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pp. 4–9. AAAI Press / The MIT Press.

Gomes, Carla P., Selman, Bart, and Kautz, Henry. 1998. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pp. 431–437. AAAI '98/IAAI '98. American Association for Artificial Intelligence.

Haken, Armin. 1985. The intractability of resolution. *Theoretical Computer Science*, **39**, 297–308.

Iwama, Kazuo. 1997. Complexity of finding short resolution proofs. In *International Symposium on Mathematical Foundations of Computer Science*, pp. 309–318. Springer.

Kaporis, Alexis C, Kirousis, Lefteris M, and Lalas, Efthimios G. 2006. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures & Algorithms*, **28**(4), 444–480.

Kautz, Henry A, Selman, Bart, et al. 1992. Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI)*, pp. 359–363. Citeseer.

Kilby, Philip, Slaney, John, Thiébaux, Sylvie, Walsh, Toby, et al. 2005. Backbones and backdoors in satisfiability. In *AAAI Conference on Artificial Intelligence*, pp. 1368–1373.

Kolaitis, Phokion G, and Vardi, Moshe Y. 2000. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, **61**(2), 302–332.

Krajíček, Jan. 2019. *Proof Complexity* vol. 170. Cambridge University Press.

Liang, Jia Hui, Ganesh, Vijay, Poupart, Pascal, and Czarnecki, Krzysztof. 2016. Learning rate based branching heuristic for SAT solvers. In Creignou, Nadia, and Le Berre, Daniel (eds), *Theory and Applications of Satisfiability Testing – SAT 2016*, pp. 123–140. Springer International Publishing.

Liang, Jia Hui, Oh, Chanseok, Mathew, Minu, Thomas, Ciza, Li, Chunxiao, and Ganesh, Vijay. 2018. Machine learning-based restart policy for CDCL SAT solvers. *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018*.

Marques-Silva, João P, and Sakallah, Karem A. 1996. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, pp. 220–227. ICCAD '96. IEEE Computer Society.

Mitchell, David, Selman, Bart, and Levesque, Hector. 1992. Hard and easy distributions of SAT problems. In *AAAI Conference on Artificial Intelligence*, pp. 1368–1373.

Moskewicz, Matthew W., Madigan, Conor F., Zhao, Ying, Zhang, Lintao, and Malik, Sharad. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference*, pp. 530–535. DAC '01. ACM.

Newsham, Zack, Ganesh, Vijay, Fischmeister, Sebastian, Audemard, Gilles, and Simon, Laurent. 2014. Impact of community structure on SAT solver performance. Sinz, Carsten, and Egly, Uwe (eds), *Theory and Applications of Satisfiability Testing – SAT 2014*, pp. 252–268. Cham: Springer International.

Pipatsrisawat, Knot, and Darwiche, Adnan. 2011. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, **175**(2), 512–525.

Tseitin, Grigori S. 1983. On the complexity of derivation in propositional calculus. *Automation of Reasoning*, pp. 466–483. Springer.

Urquhart, Alasdair. 1987. Hard examples for resolution. *Journal of the ACM (JACM)*, **34**(1), 209–219.

Vardi, Moshe Y. 2014. Boolean satisfiability: Theory and engineering. *Communications of the ACM*, **57**(3), 5–5.

Williams, Ryan, Gomes, Carla, and Selman, Bart. 2003. Backdoors to typical case complexity. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1173–1178.

Zulkoski, Edward, Martins, Ruben, Wintersteiger, Christoph M., Liang, Jia Hui, Czarnecki, Krzysztof, and Ganesh, Vijay. 2018a. The effect of structural measures and merges on SAT solver performance. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018*, pp. 436–452.

Zulkoski, Edward, Martins, Ruben, Wintersteiger, Christoph M., Robere, Robert, Liang, Jia Hui, Czarnecki, Krzysztof, and Ganesh, Vijay. 2018b. Learning-sensitive backdoors with restarts. *International Conference on Principles and Practice of Constraint Programming*, pp. 453–469. Springer.