DOI:10.1145/3560469

Today's powerful, robust SAT solvers have become primary tools for solving hard computational problems.

BY JOHANNES K. FICHTE, DANIEL LE BERRE, MARKUS HECHER, AND STEFAN SZEIDER

The Silent (R)evolution of SAT

THE PROPOSITIONAL SATISFIABILITY problem (SAT) was the first to be shown NP-complete by Cook and Levin. SAT remained the embodiment of theoretical worst-case hardness. However, in stark contrast to its theoretical hardness, SAT has emerged as a central target problem for efficiently solving a wide variety of computational problems. SAT solving technology has continuously advanced since a breakthrough around the millennium, which catapulted practical SAT solving ahead by orders of magnitudes. Today, the many flavors of SAT technology can be found in all areas of technological innovation.

SAT asks whether a given propositional formula is *satisfiable*. That is, can we set the formula's variables to values 1 (True) or 0 (False) in such a way that the entire formula evaluates to 1? $F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_2 \vee x_3)$ is a simple propositional formula in conjunctive normal form (CNF), where x_1, x_2, and x_3 are propositional variables and $\vee$, $\wedge$, and $\neg$ refer to the logical operators OR (disjunction), AND (conjunction), and NOT (negation), respectively. A variable x_i or a negated variable $\neg x_i$ is a *literal*, and a disjunction of literals is a clause. So, the above formula F is a conjunction of four clauses. The formula is satisfiable; we can satisfy it by the truth assignment that sets x_1 and x_2 to 1, and x_3 to 0: the first, third, and fourth clauses are satisfied by $x_2 = 1$ because the clauses contain x_2. The second clause is satisfied by $x_3 = 0$ because it contains $\neg x_3$. In consequence, all clauses are satisfied. A truth assignment naturally extends from variables to literals by setting $\neg x$ to the opposite value of x. Hence, a formula is satisfiable if and only if there is a truth assignment that sets at least one literal in each clause to 1.

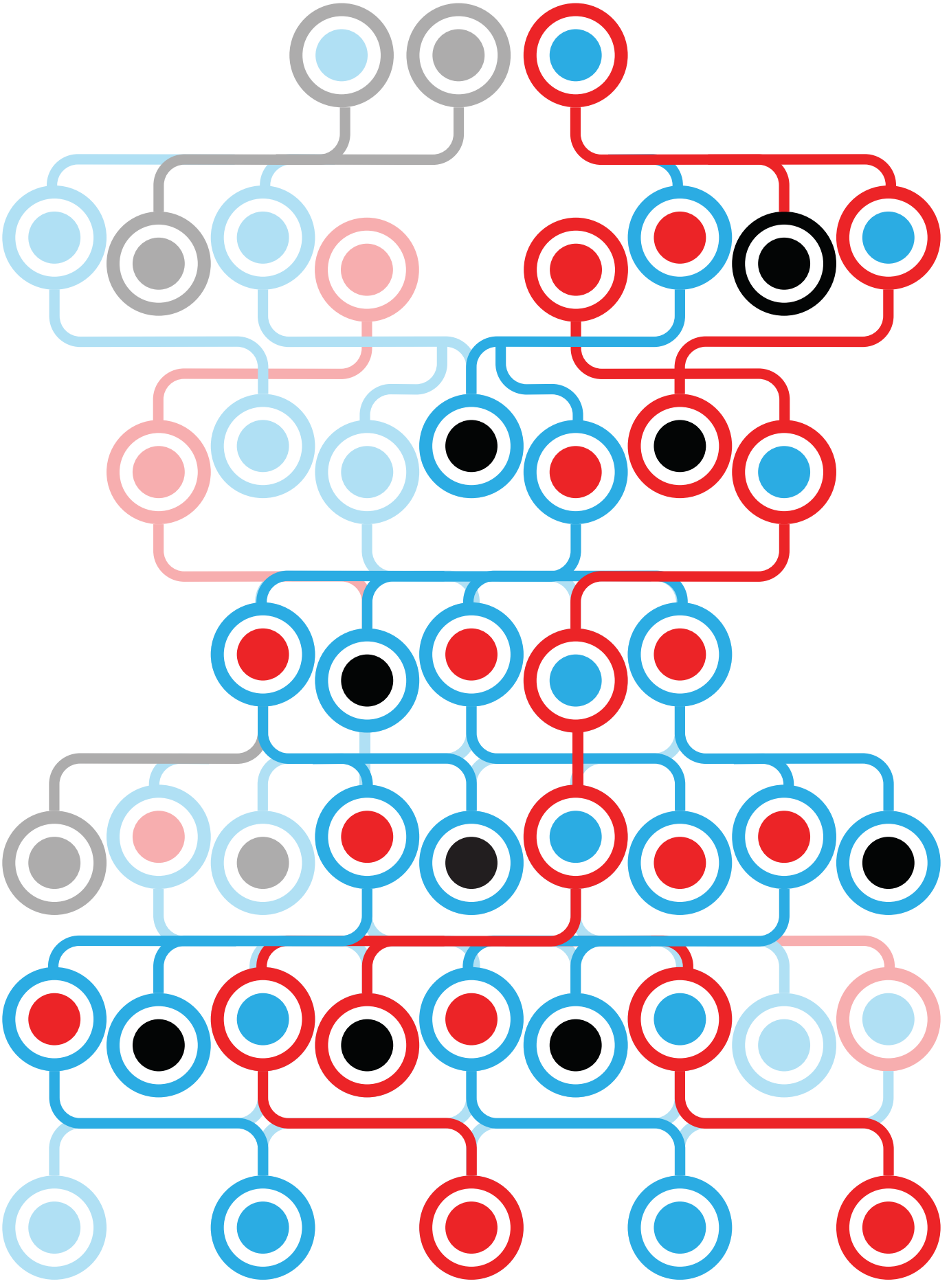
Example 1 shows a larger formula that is *unsatisfiable*—that is, not satisfied by any assignment. The focus on CNF formulas is not a restriction. The so-called Tseitin transformation[39] efficiently transforms any propositional formula into CNF without affecting its satisfiability.

At first glance, the SAT problem looks inconspicuous since it is simple to state, does not look difficult to solve, and seems uninteresting for practical purposes. Still, Stephen Cook[7] and Leonid Levin[29] showed independently in the 1970s that SAT is NP-complete, making it the first NP-complete problem. So, suppose one could solve SAT in polynomial time on arbitrary input. In

» key insights

- **Propositional Satisfiability (SAT) has been a cornerstone of computational complexity theory; now, it has become a central target problem for solving hard computational problems in practice.**
- **Since the revolution in SAT solving for decision problems that took place around the millennium, significant efficiency improvements have been achieved, and new methods for certification and trust have been added.**
- **Over the last 10 years, SAT further evolved by broadening its applications, including optimization, counting, and even problems involving quantifiers.**

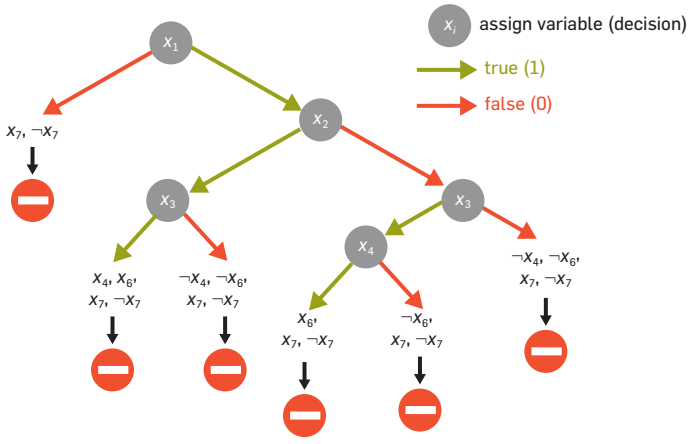
IMAGE BY IGOR KISSELEV



Example 1. Under each of the 2^9 truth assignments to the variables $x_1, \ldots, x_9$, at least one of G's clauses evaluates to 0, making the formula G unsatisfiable.*

$$G = \overbrace{(x_1 \vee \neg x_2)}^{c_1} \wedge \overbrace{(x_1 \vee \neg x_3)}^{c_2} \wedge \overbrace{(x_1 \vee \neg x_8)}^{c_3} \wedge \overbrace{(x_1 \vee \neg x_7)}^{c_4} \wedge$$
$$\overbrace{(\neg x_4 \vee x_3)}^{c_5} \wedge \overbrace{(\neg x_5 \vee \neg x_6)}^{c_6} \wedge \overbrace{(\neg x_2 \vee \neg x_3 \vee x_4)}^{c_7} \wedge$$
$$\overbrace{(\neg x_4 \vee x_6 \vee x_9)}^{c_8} \wedge \overbrace{(\neg x_3 \vee \neg x_7 \vee x_8)}^{c_9} \wedge \overbrace{(x_1 \vee x_8)}^{c_{10}} \wedge \overbrace{(x_1 \vee x_7)}^{c_{11}} \wedge$$
$$\overbrace{(x_4 \vee x_5 \vee \neg x_1)}^{c_{12}} \wedge \overbrace{(x_4 \vee \neg x_6 \vee \neg x_1)}^{c_{13}} \wedge \overbrace{(\neg x_6 \vee \neg x_4 \vee \neg x_1)}^{c_{14}} \wedge$$
$$\overbrace{(x_6 \vee \neg x_7 \vee \neg x_1)}^{c_{15}} \wedge \overbrace{(\neg x_7 \vee x_6 \vee \neg x_1)}^{c_{16}} \wedge \overbrace{(x_7 \vee x_6 \vee \neg x_1)}^{c_{17}} \wedge$$

Example 2. Search Tree: Illustration of a possible run of the DPLL algorithm on the formula *G* from Example 1. Circles indicate decision variables, and unit propagation is represented by the list of propagated literals. After obtaining inconsistency (⛔) in one branch, DPLL chronologically backtracks to the last decision, which in our example causes the search to run into the same conflict several times.



that case, one could also solve any NP-complete problem in polynomial time, and it would follow that P equals NP. Thus, in terms of worst-case complexity theory, SAT embodies computational hardness. Also, in modern complexity theory, SAT continues to serve as a hard benchmark problem in the form of the (Strong) Exponential Time Hypothesis.[4]

In stark contrast to its theoretical worst-case hardness, the SAT problem has emerged as an essential instrument for efficiently solving a wide variety of computational problems, ranging from hardware and software verification to planning, combinatorial design, and software dependency.[1,4] In this way, SAT significantly impacts today's technological innovation. SAT is widely applied in knowledge representation, reasoning,[19] and artificial intelligence (AI).[9] Although SAT is mainly associated with symbolic AI, it contributes to non-symbolic AI by providing model-counting algorithms, which are essential tools for probabilistic reasoning,[9] and allocates a main backbone for neurosymbolic AI.[24] Notably, using SAT, long-standing open problems in mathematical combinatorics were successfully solved—for example, the Pythagorean Triples Problem.[4,21] Solving such challenging problems with SAT requires the ability to efficiently translate the original problem into an instance of the SAT problem and the availability of computer programs, called *SAT solvers*, which efficiently evaluate SAT instances. Initial progress in practical SAT solving began in the early 1990s, leading to a breakthrough around the millennium. The last two decades have brought further enormous technological advancement and innovation to SAT solving. Today, SAT solvers are so powerful and robust that they have become primary tools for solving hard computational problems. Solvers have been embedded into complex procedures to solve more complex problems, such as optimization problems (MaxSAT, Pseudo-Boolean Optimization) or quantified satisfiability (QSAT).

"The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics," writes Donald E. Knuth in the preface to the second part of the fourth volume of *The Art of Computer Programming*,[28] which contains a section on satisfiability that stretches well over 300 pages.

As we observed incredible advances in computer hardware, yielding ever-faster processors, large and efficient memory, and massively parallel computing units, one could ask whether progress in SAT solving is the sole result of hardware advancement. A recent *Time Leap Challenge* addressed this question by running a race between 20-year-old SAT solvers on new computer hardware and modern SAT solvers on 20-year-old computer hardware.[16] The experiments confirm Knuth's statement on engineering and mathematics. Although hardware improvements make old solvers faster, algorithmic progress dominates and drives today's SAT solving.

This article aims to shed light on the continuing progress of practical SAT solving through a series of evolutionary innovations and improvements that have been ongoing since the revolutionary breakthrough around the millennium. Overall, we argue that SAT has earned the title of a *silent (r)evolution*. We tell the story of SAT divided into three eras: the pre-revolution, the revolution, and the evolution.

Eras of Practical SAT Solving

Era I: The pre-revolution. The building blocks of today's SAT success are various and date back even to the first half

of the 20th century. We refer to other sources for a detailed description of SAT's history[4] and focus on a few important milestones of the modern era. Complete and incomplete solvers were the prevalent SAT solvers in the 1990s. *Incomplete solvers*, based on stochastic local search,[23] were successfully applied to planning problems[4] and satisfiable instances. In contrast, *complete solvers*, based on backtracking search, were used to solve combinatorial problems (such as puzzles, N-queens, and Latin squares) as well as uniform random k-SAT formulas (including unsatisfiable ones). These early complete solvers followed the general approach, called *DPLL*, which was first proposed by Davis, Logemann, and Loveland (DLL) as a memory-efficient refinement of an earlier algorithm by Davis and Putnam (DP).[a,11,12]

In modern terminology, DPLL is a backtracking algorithm that performs a depth-first exploration of a binary search tree on truth assignments (see Example 2). It applies the following two optimization steps repeatedly after a variable has been assigned, propagating the current partial assignment: if the current assignment sets all but one literal of a clause to 0 (the clause is a unit clause), then we can safely set the remaining unassigned literal to 1—*unit propagation* is the repeated application of this process; if the current assignment satisfies all clauses containing some uassigned literal, then we can safely set that literal to 0—the opposite literal is called a *pure literal*. Backtracking occurs when the current partial assignment sets all the literals of a clause to 0.

Heuristic methods that decide when and how variables are assigned play a crucial role in the context of DPLL.[8]

Early implementation challenges. Three SAT competitions were organized in the 1990s. The second competition, 1993's DIMACS implementation challenge,[4] introduced the standard ASCII input format for SAT solvers, which is still in use. This standardized input format supports reusing SAT solvers as black boxes and sharing benchmarks. The DIMACS CNF format describes a propositional formula. The preamble provides the format (CNF), the number of variables, and the number of clauses. The remaining lines denote clauses, where each literal is represented by a signed integer, using 0 as a separator. Example 3 illustrates our running example containing nine variables and 17 clauses in the DIMACS CNF format.

Era II: The revolution. The first pillar of the revolution[30] was the solver GRASP,[31] which proposed a new architecture, combining non-chronological backtracking, conflict analysis, and learning, today referred to as *conflict driven clause learning (CDCL)*. CDCL is more than just DPLL with learning, since it is no longer a pure backtracking search. It captures unit propagation in a directed acyclic implication graph that is used to perform conflict analysis and clause learning, which prominently drives the search. Modern SAT solvers use the *trail* data structure proposed by MiniSat[13] to capture the search tree and the implication graph.

The second pillar was the CDCL-based solver Chaff,[32] specially designed to solve large benchmark instances by taking the characteristics of the host computer into account and achieving an unprecedented balance between the sophistication of algorithms and data structures on the one side and the practical efficiency on the other. Chaff introduced the Watched Literal data structure, a "lazy" scheme for performing unit propagation that allowed cost-free backtracking. The branching heuristics (VSIDS) and conflict analysis procedure were carefully designed with a new tradeoff between reasoning power and computation cost: The emphasis on recent conflicts leads to a locality-based search.

We illustrate CDCL for our running example in Example 4. Many conflict clauses are learned during the search, which is quite demanding on memory. Modern CDCL solvers frequently delete learned clauses and heuristically predict the ones to keep.[4] Various heuristics, such as VSIDS,[32] EVSIDS,[13] or LRB,[4] are available for optimizing the search and assigning the variables. Those heuristics impose low overhead, generate almost no cost for backtracking, and are updated when learning clauses. To escape unlucky search tree exploration during the search, solvers frequently unassign all variables and restart the search (but keep the learned clauses). Common heuristic schemes that decide when to restart are Luby or Rapid Restarts.[4] Also, machine-learning techniques have been used to optimize heuristics.[4] Still, more careful analysis is needed to better understand how the components of the solver and their interaction affect its overall efficiency.[14,27]

Some critical applications drew much attention to practical SAT solving from academia and industry. For example, bounded model checking (BMC)[4] provided numerous challenging benchmarks, which were soon tackled. BMC, which checks the correctness of sequential systems over a bounded number of steps, was proposed after the failure of standard model checking with binary decision diagrams (BDDs), a graph-based canonical representation of propositional formulas.[6] The BMC benchmarks were extensive, with tens of thousands of variables and clauses. Combining conflict-driven heuristics with conflict analysis (CDCL) could solve BMC benchmarks up to two orders of magnitude faster than any other approach. These impressive results applied to other application benchmarks as well. The computer-aided verification (CAV) community acknowledged the central role of GRASP and Chaff for that outstanding progress by presenting their authors with the CAV 2009 award for "fundamental contributions to the development of high-performance Boolean satisfiability solvers."

Era III: The evolution. While improving solvers is an important goal, the community has progressed in new modeling techniques and methods for certifying results.

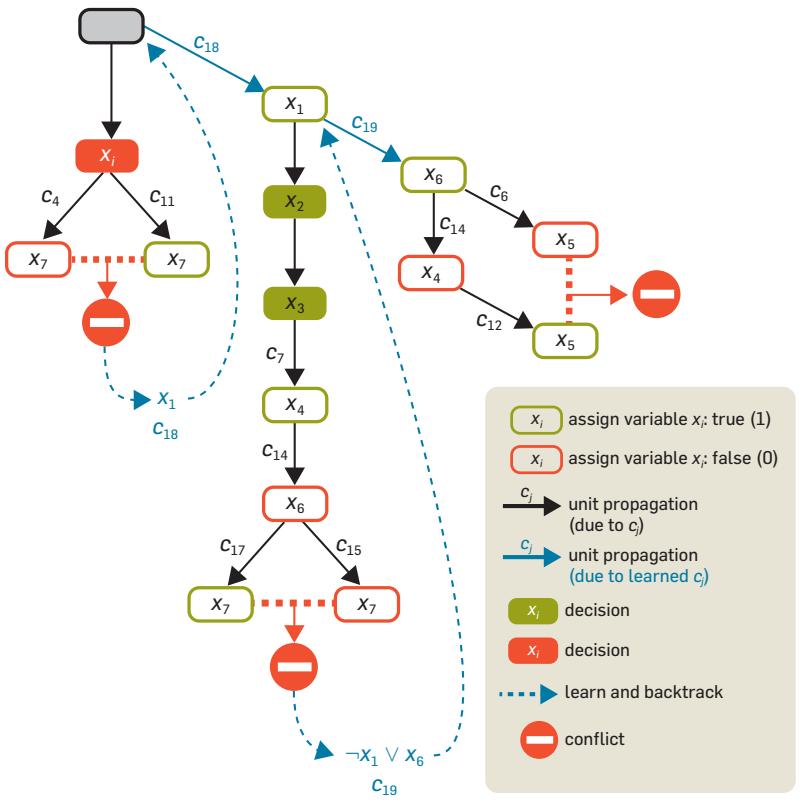
Efficient SAT encodings. It is challenging to express a computational problem as a CNF formula that the solver can solve efficiently. Encodings may blow up the resulting CNF formula quite notably, resulting in quadratic, cubic, or even

Example 3. DIMACS CNF representation of Example 1's formula *G*. Try this example at http://bit.ly/406Lqc7 using for instance kissat SAT solver http://bit.ly/3oa8zx9 to solve it with a real SAT solver.

```
p cnf 9 17    -5 -6 0       4  5-1 0
1 -2 0        -2 -3 4 0     4 -6-1 0
1 -3 0        -4  6 9 0    -6 -4-1 0
1 -8 0        -3 -7 8 0     6 -7-1 0
1 -7 0         1  8 0      -7  6-1 0
-4  3 0        1  7 0       7  6-1 0
```

a DPLL combines DP and DLL to emphasize both contributions.

Example 4. Propagation Graph: Trace of a possible CDCL run on the formula *G* from Example 1.



Boxes represent assigning a truth value to a variable. Filled boxes represent decisions. Edges mark the underlying reason, which can be a decision (no label) or unit propagation (clause ID label). First, we decide that x_1 is set to 0. By conflict analysis of the clauses causing the conflict (c_4 and c_{11}), we immediately learn the conflict clause $c_{18} = x_1$. Then, we backtrack and after unit propagation, we perform two decisions on variables x_2 and x_3, which allows us to finally learn the conflict clause $c_{19} = \neg x_1 \vee x_6$. This additional clause, together with unit propagation, already results in the conclusion that *G* is unsatisfiable, and there are no decisions left for backtracking. In contrast to Example 2, CDCL applies non-chronological backtracking and learning to avoid rediscovering the same conflict several times, by learning—for instance—that when x_1 is set to 1, x_6 has to be set to 1 as well (clause c_{19}). We observe that it allowed us to reduce the number of decisions from five to three, and the number of conflicts from six to three.

Example 5. Encoding techniques based on binary adders.[37]

Let *H* be a formula that contains many variables, say $v_1, v_2, \ldots, v_{1000}$. Assume that for modeling purposes, we are interested in restricting the formula in such a way that at most one of these variables is allowed to be set to 1. We can achieve this by adding all the $\binom{1000}{2} = 499{,}500$ binary clauses $\neg v_i \vee \neg v_j$, $1 \le i < j \le 1{,}000$; this generates a huge formula. However, we can avoid this quadratic space requirement and express the restriction much more succinctly with only a linear overhead. Namely, we can mimic well-known logic circuits that form binary adders with carryover bits in terms of clauses, involving the original variables v_i and additional auxiliary variables.

worse overhead compared to the original problem instance. Usually, one faces a tradeoff between space and simplicity. Intuitively, one would try to keep the instance as small and concise as possible, balancing the number of variables and the number of clauses. If we have n variables and add m auxiliary variables, the potential search space grows from 2^n to 2^{n+m}. Still, in practice, adding such variables can reduce the number of clauses, thereby speeding up the search.[4] A similar pattern was observed by early automatic test pattern-generation programs, such as path-oriented decision making (PODEM).[4] This is particularly relevant when encoding properties that constrain the number of variables from a set of variables that must be set to 1, called *cardinality constraints*. We give a simple example in Example 5.

Assumptions. Using SAT solvers as "oracles" is far more complex in practice than just having an NP oracle in theoretical considerations. On the one hand, as successful as SAT solvers are, they are not NP oracles. SAT solvers need time for solving and may not even answer within a reasonable time. On the other hand, modern SAT solvers are more sophisticated than NP oracles. They admit *stateful* procedures, where clauses may be added or removed during solving. To control added clauses without invalidating learned clauses, modern solvers support *assumptions*, popularized by MiniSat,[13] which are additional literals introduced for representing the state of clauses. It turned out that assumptions are highly versatile and useful. We demonstrate assumptions in Example 6.

Inprocessing/preprocessing. SAT solvers run various simplification rules to eliminate unnecessary variables or clauses. Several simplification rules exist; some can be carried out in linear time while others require quadratic time. Extremely beneficial is preprocessing, where the instance is simplified before the actual solver is started.[4] Interestingly, one such preprocessing task is to run a restricted form of the Davis-Putnam procedure on a subset of variables (bounded variable elimination). However, modern solvers such as Lingeling implement more sophisticated rules, which are run during the search before the next variables are assigned (inprocessing) and take learned clauses into account.[4] More time-consuming inprocessing techniques such as equivalence learning, cardinality constraints, or parity detection can be applied with a limited budget and interleaved with search.

Parallel solving. In the 2000s, Moore's law started approaching its limits in CPU performance on single cores, and the clock-speed improvements for silicon-based chips started to slow down. Parallel computation was one way to compensate for this trend, and it found its way into consumer hardware. Around the same time, parallel computing also started to become more fashionable in SAT solving, aiming to improve overall processing performance and tackle huge instances.[21] The two main techniques

that evolved are *parallel-portfolio* and *divide-and-conquer* solving.

In parallel-portfolio solving, a problem instance is independently given to a collection of solvers competing for a solution in parallel. Usually, the collection comprises different solvers or one solver with different heuristics. While parallel SAT solvers optimize performance to solve large instances, they may produce different satisfying assignments or unsatisfiability proofs for a given input formula because they are inherently non-deterministic due to race conditions. Nevertheless, some recent parallel solvers achieve determinism with a reasonable runtime overhead. Determinism is critical for applications requiring reproducibility, such as scheduling or model checking. Over the last years, massive parallelization using graphics processing units (GPUs) or tensor processing units (TPUs) has become a popular approach in computer science and ML. Aspects of modern SAT solving have been successfully implemented and tested on GPUs, including solvers tailored for counting the number of satisfying assignments.[17] Still, SAT solving has not yet benefited from massive parallelization.

Divide-and-conquer divides a formula into multiple formulas. The workload is shared among multiple solvers that run in parallel and can even synchronize learned clauses. However, modern sequential SAT solvers do not need to exhaust the search space to find a solution or refute a formula; learned clauses allow for shortcuts, making it hard to decompose instances because of the increased synchronization efforts for learned clauses. A popular version is *cube-and-conquer*, which partitions an instance by a look-ahead technique into up to a million sub-instances and then solves the sub-instances with a CDCL-based solver.[21]

Correctness of solvers and emitting proofs. The results of early SAT solvers were hard to validate. While it is easy to check whether an assignment returned by a solver satisfies the given formula, in the past, no technique was available to verify correctness if the solver claimed that an instance is unsatisfiable. During the first competitions, one could only check whether emitted assignments were correct and that solvers did not provide inconsistent results. If solver A outputs "satisfiable" and solver B outputs "unsatisfiable" on the same formula, one would check whether the assignment given by A indeed satisfies the formula. If so, solver B is incorrect; otherwise, solver A is incorrect, but nothing can be concluded about solver B. Solver correctness has been addressed by intensive testing, including fuzzing and asking for traces that give rise to an unsatisfiability proof within a dedicated propositional proof system.[4] Such traces can be automatically checked using third-party tools, which can be specified, implemented, and fully verified. The relationship between CDCL runs and propositional proof systems is now well understood. All the relevant proof systems rely on the *resolution rule*, which derives a new clause from two clauses containing exactly one opposite literal. For instance, in Example 4, the clause $c_{18} = x_1$ can be derived by resolution from $c_4 = x_1 \vee \neg x_7$ and $c_{11} = x_1 \vee x_7$; clause $\neg x_1 \vee x_6$ can be derived by resolution from $c_{17} = x_7 \vee x_6 \vee \neg x_1$ and $c_{15} = x_6 \vee \neg x_7 \vee \neg x_1$. In theory, CDCL-solvers follow resolution proofs that reuse derived clauses (DAG-like resolution).[3,34] They can be exponentially more succinct than tree-like resolution proofs, corresponding to DPLL-solvers.[4] In the early 2000s, techniques to ensure the correctness of unsatisfiability results using variants of resolution proofs were presented.[4] However, they were invasive to the

Example 6. Assumptions make our running formula satisfiable again.

Think of a situation where a SAT solver has already solved a formula—for example, formula G from Example 1. Now, assume that the application from which the formula arose changes, and suddenly clause c_{17} is no longer present. Instead of solving the modified formula from scratch, SAT solvers can more efficiently handle such changes through *assumptions*. We can imagine the solver keeping a special assumption literal in each clause that indicates whether the clause should be disregarded. By default, these literals are assumed false, and therefore all clauses need to be satisfied. Typically, the SAT solver does not terminate after solving a formula but keeps its state and learned clauses. When we remove clause c_{17} from G, the assumption literal corresponding to c_{17} flips to true, which deactivates the clause. Since assumption literals are internally also considered for learned clauses, the learned clause c_{19} implicitly depends on c_{17}. Retracting c_{17} deactivates the learned clause, as it is already satisfied by the assumption literal. From Example 4, we can see that the conflict in the middle disappears without clause c_{17}. Instead, we can immediately construct a satisfying assignment. Thus, the already-spent effort on solving can be fruitfully used without re-solving the entire formula again, similar to how an experienced human proceeds when updating the search procedure with a pen on paper.

Example 7. Illustration of a DRAT proof for our running example.

L1: a 1 0
L2: a –1 6 0

$G_{L1} = G \wedge (x_1)$
$G^R_{L2} = G_{L1} \wedge \neg c_{19}$
$= G_{L1} \wedge \neg(\neg x_1 \vee x_6)$
$= G_{L1} \wedge x_1 \wedge \neg x_6$

$G_{L2} = G_{L1} \wedge (\neg x_1 \vee x_6)$

(a) Output of a trace that corresponds to the learned clauses in Example 1. We first add the clause $c_{18} = (x_1)$ by the line L1, then we add the clause $c_{19} = (\neg x_1 \vee x_6)$ by the line L2, which concludes the run after unit propagation, determining unsatisfiability. Note that when each of the clauses is added, it is *Reverse Unit Propagatable (RUP)*—that is, adding the unit clauses corresponding to the negation of the clause's literals would allow unit propagation to determine the formula unsatisfiable.

(b) After line L1 in Example 7a (left), we have already added c_{18}—that is, we consider G_{L1}. If, now at line L2, we invert the clause c_{19} by negating it, which is also referred to as *reversing* the clause, G^R_{L2} s determined as unsatisfiable by unit propagation.

(c) After line L2 in Example 7a (left), we obtain G_{L2}, which is determined as unsatisfiable just by applying unit propagation. We can see this also in the rightmost part of Example 4. There, no further decisions are required for obtaining the last conflict. Indeed, one could also simplify the proof and add only the clause x_6 in line 2 instead of c_{19}. Such compressions are widely used and speed up the solving process.

solver and produced enormous proofs. In the 2010s, techniques emerged that provide proofs that are compact, easy to emit, can be checked efficiently, and offer high expressiveness. One popular proof format is Deletion and Resolution Asymmetric Tautology (DRAT).[4] Example 7a provides a DRAT proof for our running example but uses only a fragment of DRAT's features.

Problem decomposition and verification. Problem decomposition and verification were a game-changer for solving long-standing open math problems using SAT technology—for example, invalidating a hypothesis on Pythagorean triples.[22] As described previously, modern SAT solvers emit proofs that cover all types of present solving techniques, including preprocessing and inprocessing. A proof of a mathematical statement such as the propositional Pythagorean triples problem can be longer than 200TB, which is more data than the Hubble Space Telescope accumulated in about 20 years. Such proofs are clearly beyond human understanding and not human-verifiable simply because of their length but can still be automatically checked. As a result, it leads to using SAT solvers as brute-force technology for mathematical applications. This progress in SAT solving fulfills, finally, some of the early promises in automated reasoning and deduction from the 1960s.

Modern competitions and open source. Since 2002, the SAT community has organized annual competitive events that encourage novel solver design and gather new benchmark instances.[26] Competitions continue to impact progress in solver development and provide a way to independently assess the efficiency of submitted solvers. In 2002, few solvers used the CDCL architecture (4 out of 19). Solvers were not robust, and many of them crashed during the competition—for example, on instances with many variables, many clauses, or very long clauses. Fortunately, this changed the following year due to various widely available benchmarks and published expected results. Initially, many solvers were closed source. Strict submission requirements and the success of the free solver MiniSat[13] shifted the community to open source. MiniSat code was simple to read, compact, and designed for extendibility. It implemented CDCL and outperformed all other solvers in 2005 due to minimizing learned clauses and the preprocessor SatELite.[4] Over time, the open source principle enabled researchers to implement their ideas on top of the best available solvers and understand crucial implementation details usually not shared in papers. As of 2023, offsprings of MiniSat, such as Glucose, MapleSAT, and MapleCOMSPS, still participate in the SAT competition. While competitions continuously raise standards in efficient solving, there is no single best solver in practice. Various solvers perform differently depending on the considered class of benchmarks or the configuration of their internal heuristics. This diversity can be exploited by systems such as SATzilla,[4] which predict the performance of existing solvers and select the most promising ones. A more sophisticated approach is automatic algorithm selection and tuning. Tools such as AUTOFOLIO or SMAC[4] find good parameters for a given solver on a target set of benchmarks.

Theoretical understanding of efficient solving. In 2014, Moshe Vardi pointed out the lack of theoretical understanding of the success in practical SAT solving.[40] Meanwhile, the understanding improved from various viewpoints.[18] Solvers are well-engineered, constantly improved, and efficiently solve many classes of real-world instances. Still, they perform poorly on certain random and cryptographic instances. A standard view is that industrial instances contain a "hidden structure" that the SAT solver manages to use implicitly. Any theoretical analysis that ignores structural properties of the SAT instance hence cannot match up with the empirically observed solver performance. This shortcoming led to several research lines aiming to capture the hidden structure in a SAT instance mathematically. One line focuses on capturing structure that correlates well with empirically observed solver performance.[2] This provides a statistical correlation but no guarantee that every SAT instance that exhibits this structure is solved quickly. Another line of research takes a causal approach; it uses the theoretical framework of parameterized complexity to obtain asymptotic performance guarantees for algorithms that explicitly detect and exploit the hidden structure.[4] The

Selected Formalisms

Selected formalisms beyond propositional logic inspired by CDCL.

SAT Modulo Theories (SMT)
SMT[4] extends SAT by theories of first-order logic, so that instances can express questions beyond CNF and propositional variables. Among the more prominent theories supported by SMT solvers are the theory of arithmetic over integers or real numbers, the theory of equality and uninterpreted functions, as well as theories over different data structures such as arrays and bit-vectors. These theories are readily available in many state-of-the-art SMT solvers, such as z3, cvc4, and MathSAT5. Initial attempts to solve SMT resulted in translating instances to propositional formulas, where a SAT solver was responsible for solving. Over time, more sophisticated techniques and architectures emerged. Still, the core of modern SMT architectures is powered by SAT engines and often referred to by CDCL(T), where T denotes the theory under consideration. The performance of current SMT solvers resulted in challenging SMT competitions, which further led to standardized collections and libraries of instances. Some of these instances indeed stem from crucial applications, mostly in the area of software verification.

Answer Set Programming (ASP)
Propositional ASP builds on top of SAT and adds concepts from knowledge representation and reasoning (stable models). It can solve problems from the second level of the polynomial hierarchy. Non-propositional ASP provides a declarative programming-oriented approach with first-order variables and extended statements toward compact and human-readable representations of encodings.[19] There, problems are NEXPTIME-hard. Still, modern ASP solvers use CDCL-based techniques. ASP became a de facto successor of Prolog. Modern ASP solvers even offer features from SMT, including ways to compactly define own theories.

Constraint Programming (CP)
CP focuses on modeling and solving combinatorial problems while offering engineers a wide variety of constraints, which are not limited to the propositional domain.[36] Furthermore, available constraints are very expressive and high-level. Surprisingly, CDCL-based solvers have a strong influence as the core engine in some solvers, as witnessed in the MiniZinc 2022 Challenge—for instance, PicatSAT and OR-Tools.[38]

Propositional Problems

A short list of selected propositional problems beyond SAT that employ CDCL.

Quantified Boolean Formulas (QBFs)
QBFs extend CNF formulas by quantifying variables universally (∀) or existentially (∃). SAT can be seen as the special case where all variables are existentially quantified. With QBFs, one can encode some problems exponentially more succinctly than with SAT, but checking whether a QBF is true (QSAT) is PSPACE complete.[4]

Pseudo-Boolean (PB) Solving
PB solving supports linear constraints, which are more expressive than clauses. A linear constraint is a linear combination of propositional variables—for example, $2x_1 + \ldots - 3x_k \leq 5$. With the use of auxiliary variables and clauses, such constraints can also be encoded in CNF, but linear constraints eliminate the need for constructions, such as in Example 5. Furthermore, one can use stronger proof systems on such instances—for example, Gaussian elimination or cutting planes.[4]

Maximum Satisfiability (MaxSAT)
One can extend CNFs by marking clauses as hard or soft.[4] A hard clause must be satisfied and a soft clause should be satisfied if possible. Then, the problem MaxSAT asks to find an assignment that satisfies all the hard clauses and a maximum number of soft clauses. Today, there are many applications where MaxSAT thrives, and the list of use cases ranges from configuration problems in industry and software management, across combinatorial optimization, to data analysis and probabilistic reliability estimation of systems.

Sampling, Counting, Compilations
Advances in SAT solving have been extended to generating multiple diverse solutions (sampling), counting the number of solutions,[4,15] and compiling a CNF formula into another normal form, such as d-DNNF, where various problems can be solved efficiently (knowledge compilation).[10] Practical applications of counting and sampling can be found in probabilistic reasoning and bioinformatics. Sampling is of particular interest in testing or performance-model optimization. Knowledge compilation and compilers are ideal technologies for various problems in decision making and AI.

underlying fixed-parameter algorithms are specific to the parameter under consideration and generally do not aim at providing a theoretical explanation of CDCL-solver performance.

The Art of Using SAT Solvers

A SAT solver can be seen as a powerful engine; however, reaching a solution or guaranteeing that there is none requires more than an engine alone. Therefore, the SAT community developed innovative and powerful methods for using and adapting SAT engines.

Single-call solving. *Static encodings* transform an input instance of the problem of interest into a CNF formula. Encoding can be accomplished in a high-level programming language such as Python, which produces a DIMACS CNF file. Slightly higher-level access can be obtained by PySAT,[25] a popular Python library for encoding problems and using state-of-the-art SAT solvers. SAT engines are highly optimized on the low-level DIMACS CNF format for squeezing out every ounce of performance. Over time, engineers have shifted to more human-readable and reusable languages; some go beyond the propositional case. Examples include the ASP input language[19] or the MiniZinc constraint modeling language.[38] The "Selected Formalisms..." sidebar on the previous page lists other formalisms that focus on modeling and problem solving with more expressive languages. Still, detailed domain knowledge and the skill of efficiently driving the engine are often helpful.

Multi-call solving. Modern SAT solvers can go beyond single-call solving. Many such solvers can modify an already-solved formula, reusing information from a previous solving process (*incremental solving*) and outputting a subset of the input clauses that remain unsatisfiable (*unsatisfiable core*). Some of these techniques resulted in efficient solving methods for problem formalisms beyond SAT, where the SAT engine drives the solving process. The "Propositional Problems" sidebar on this page lists such selected formalisms at the propositional level.

SAT solvers as subroutines. Many hard computational problems of practical significance, particularly in AI, are beyond NP. An unexpectedly successful application of SAT solvers comes in terms of QBF solvers for the QSAT problem ("Propositional Problems" sidebar). Several successful QBF solvers, such as CAQE and RAReQS,[4] employ SAT solvers. RAReQS uses two interacting SAT solvers internally to determine the validity of the QSAT instance. Each solver acts as the existential or the universal player in a two-player game. The SAT solvers drive the procedure: The models found refine the abstraction of the players. This generic approach is called *Counter Example Guided Abstraction Refinement* (CEGAR). Other successful QBF solvers, such as DepQBF[4] and Qute,[33] extend the CDCL procedure to the quantified case, with special methods for identifying and dealing with the dependencies between variables imposed by the quantifiers. In hardware or software verification, the model-checking tool IC3[5] goes further than finding bugs by proving safety properties. IC3 uses incremental SAT to check the reachability of states from both the initial states and the states of interest. A different flavor of multi-call solving can be used for optimization problems, which are too large to be encoded into a single SAT or MaxSAT instance and where optimal solutions are out of reach. The *SAT-based Local Improvement Method* (SLIM) starts with an initial solution provided with some standard heuristics. Then, SLIM repeatedly replaces local parts with improved solutions obtained by a SAT or MaxSAT solver. The method has been successfully applied for graph decomposition, decision-tree induction, and Bayesian Network structure learning.[35]

Core-guided optimization. SAT solvers can also solve optimization problems for minimizing or maximizing an objective function. For instance, such a function could count the variables from a subset of variables set to 1. Incremental solving provides the means to optimize with a SAT solver: The solver is used by progressively adding more and more clauses that further constrain the instance until it becomes unsatisfiable. Recall that a SAT solver can provide for an unsatisfiable instance with an unsatisfiable core.[4] To satisfy the maximum number of clauses, at least one clause from the unsatisfiable core needs to be disregarded. This can be expressed using assumptions, a technique referred

Example 8. Soft clauses allow us to deal with inconsistent requirements.

Recall Example 6, where we introduced an assumption on clause c_{17}. Instead, we could also declare clause c_{17} a *soft clause* that should be satisfied if possible, not mandatorily. From our example, we can see that this clause cannot be satisfied, as the entire formula with c_{17} is unsatisfiable.

to as *core-guided optimization*. Notably, core-guided optimization is very successful when solving problems such as MaxSAT, the problem of maximizing the number of satisfied clauses (Propositional Problems sidebar). Example 8 describes the use of soft clauses on our running example, which is the basis for MaxSAT. A popular approach for MaxSAT solving uses as a subroutine the computation of a minimal hitting set on unsatisfiable cores, usually achieved using a Mixed-Integer Programming (MIP) solver.

Outlook

Over the last two decades, SAT solving techniques have changed how we tackle hard computational problems. The SAT revolution is significantly less known than the celebrated success of machine learning with its ubiquitous and widely reported impact on technology and society. SAT solvers have influenced modern technology more silently. They are used in computational biology,[20] for planning,[4] to verify modern hardware,[4] operating systems, software,[4] and even mathematical statements.[4] This makes SAT crucial to the progress of modern information technology. Still, critical Computer Science challenges related to SAT solving are still ahead: How can we further improve parallel search to take full advantage of modern massively parallel hardware? Why does SAT solving often work so well in practice, and what characterizes the cases where it struggles? How can we improve the process of coming up with good encodings? Finally, will SAT be widely applied also to computational physics, chemistry, or non-symbolic AI? To some extent, the revolutions of SAT and machine learning are complementary. There is much potential in combining the two.

Acknowledgments

This work was carried out while the authors visited the Simons Institute for the Theory of Computing. It has been supported by a Google Fellowship at the Simons Institute; the Austrian Science Fund (FWF); Grants J4656, Y698, and P32830; and the Vienna Science and Technology Fund, Grant WWTF ICT19-065.

References

1. Abate, P. et al. Dependency solving: A separate concern in component evolution management. *J. Systems and Software 85*, 10 (2012), 2228–2240.
2. Ansótegui, C. et al. Community structure in industrial SAT instances. *J. Artificial Intelligence Research 66* (2019), 443–472.
3. Atserias, A., Fichte, J.K., and Thurley, M. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artificial Intelligence Research 40*, 1 (2011), 353–373.
4. A. Biere, M. Heule, H. van Maaren, and T. Walsh, (Eds.). *Handbook of Satisfiability (2nd Edition)*. IOS Press, Amsterdam, Netherlands (2021).
5. Bradley, A.R. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation–12th Intern. Conf. Proceedings (Lecture Notes in Computer Science 6538)*, R. Jhala and D.A. Schmidt (Eds.). Springer (January 2011), 70–87.
6. Bryant, R.E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions in Computing* C-35, 8 (August 1986), 677–691.
7. Cook, S.A. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual Symp. on Theory of Computing*, ACM (1971), M.A. Harrison, R.B. Banerji, and J.D. Ullman (Eds.) 151–158.
8. Cook, S.A. and Mitchell, D.G. Finding hard instances of satisfiability problem: A survey. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science 5* (1997).
9. Darwiche, A. Three modern roles for logic in AI. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems* (2020), 229–243.
10. Darwiche, A. and Marquis, P. A knowledge compilation map. *J. Artificial Intelligence Research 17*, 1 (2002), 229–264.
11. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Communications of the ACM 5*, 7 (July 1962), 394–397.
12. Davis, M. and Putnam, H. A computing procedure for quantification theory. *J. of the ACM 7*, 3 (1960), 201–215.
13. Eén, N. and Sörensson, N. An extensible SAT-solver. In *Proceedings of the 6th Intern. Conf. on Theory and Applications of Satisfiability Testing*, E. Giunchiglia and A. Tacchella (Eds.), Springer Verlag (2003), 502–518.
14. Elffers, J. et al. Seeking practical CDCL insights from theoretical SAT benchmarks. In *Proceedings of the 27th Intern. Joint Conf. on Artificial Intelligence*, J. Lang (Ed.), (2018), 1300–1308.
15. Fichte, J.K., Hecher, M., and Hamiti, F. The model counting competition 2020. *ACM J. Experimental Algorithmics 26*, 13 (December 2021).
16. Fichte, J.K., Hecher, M., and Szeider, S. A time leap challenge for SAT-solving. In *Proceedings of the 26th Intern. Conf. on Principles and Practice of Constraint Programming*, H.Simonis (Ed.). Springer Verlag, Louvain-la-Neuve, Belgium (2020), 267–285.
17. Fichte, J.K., Hecher, M., and Zisser, M. An improved GPU-based SAT model counter. In *Proceedings of the 25th Intern. Conf. on Principles and Practice of Constraint Programming*, T. Schiex and S. de Givry (Eds.), (2019), 491–509.
18. Ganesh, V. and Vardi, M.Y. *Beyond the Worst-Case Analysis of Algorithms*. Cambridge University Press, (2021), 547–566.
19. Gebser, M. et al. *Answer Set Solving in Practice*. Morgan & Claypool (2012).
20. Guerra, J. and Lynce, I. Reasoning over biological networks using maximum satisfiability. In *Proceedings of the 18th Intern. Conf. Principles and Practice of Constraint Programming (Lecture Notes in Computer Science 7514)*, M. Milano (Ed.). Springer Verlag, Québec City, QC, Canada (2012), 941–956.
21. Hamadi, Y. and Sais, L. (Eds.). *Handbook of Parallel Constraint Reasoning*. Springer (2018).
22. Heule, M.J.H. and Kullmann, O. The science of brute force. *Communications of the ACM 60*, 8 (July 2017), 70–79.
23. Hoos, H.H. and Stützle, T. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence 112*, 1 (1999), 213–232.
24. Ignatiev, A. et al. Reasoning-based learning of interpretable ML models. In *Proceedings of the 30th Intern. Joint Conf. on Artificial Intelligence*, Zhi-Hua Zhou (Ed.), (August 2021), 4458–4465.
25. Ignatiev, A., Morgado, A., and Marques-Silva, J. PySAT: A Python toolkit for prototyping with SAT oracles. In *Proceedings of the 21st Intern. Conf. on Theory and Applications of Satisfiability Testing (Lecture Notes in Computer Science 10929)*, O. Beyersdorff and C.M. Wintersteiger (Eds.). Springer Verlag (2018), 428–437.
26. Järvisalo, M. et al. The International SAT Solver Competitions. *AI Magazine*. The AAAI Press (2012).
27. Katebi, H., Sakallah, K.A., and Marques-Silva, J.P. Empirical study of the anatomy of modern SAT solvers. In *Proceedings of the 14th Intern. Conf. on Theory and Applications of Satisfiability Testing (Lecture Notes in Computer Science 6695)*, K.A. Sakallah and L. Simon (Eds.). Springer Verlag (2011), 343–356.
28. Knuth, D.E. *The Art of Computer Programming Vol. 4B*, Combinatorial Algorithms, Part 2. Addison-Wesley (2023).
29. Levin, L. Universal sequential search problems. *Problems of Information Transmission 9*, 3 (1973), 265—266.
30. Malik, S. and Zhang, L. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM 52*, 8 (August 2009), 76–82.
31. Marques-Silva, J. and Sakallah, K.A. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Comput. 48*, 5 (May 1999), 506–521.
32. Moskewicz, M.W. et al. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conf.*, J. Rabaey (Ed.). Association for Computing Machinery (2001), 530–535.
33. Peitl, T., Slivovsky, F., and Szeider, S. Dependency learning for QBF. *J. Artificial Intelligence Research 65* (2019), 180–208.
34. Pipatsrisawat, K. and Darwiche, A. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence 175*, 2 (2011), 512–525.
35. Ramaswamy, V.P. and Szeider, S. Turbocharging treewidth-bounded Bayesian network structure learning. In *Proceedings of the 35th AAAI Conf. on Artificial Intelligence*, The AAAI Press (2021), 3895–3903.
36. Rossi, F., van Beek, P., and Walsh, T. *Handbook of Constraint Programming*. Elsevier Science Publishers, North-Holland, USA (2006).
37. Sinz, C. Towards an optimal CNF encoding of Boolean cardinality constraints. In *Proceedings of the 11th Intern. Conf. on Principles and Practice of Constraint Programming (Lecture Notes in Computer Science 3709)*, P. van Beek (Ed.). Springer Verlag, Sitges, Spain (2005), 827– 831.
38. Stuckey, P.J. et al. The MiniZinc Challenge 2008–2013. *AI Magazine 35*, 2 (June 2014), 55–60.
39. Tseytin, G.S. On the complexity of derivation in propositional calculus. *Automation of Reasoning: Classical Papers in Computational Logic 2* (1983), 466–483.
40. Vardi, M.Y. Boolean satisfiability: Theory and engineering. *Communications of the ACM 57*, 3 (March 2014), 5.

Johannes K. Fichte is an associate professor at IDA, Institutionen för datavetenskap, Linköping University, Sweden.

Daniel Le Berre is a professor at Artois University and CNRS, Centre de Recherche en Informatique de Lens, France.

Markus Hecher (hecher@mit.edu) is a PostDoc at the Computer Science & Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, USA.

Stefan Szeider is a professor and head of the Algorithms and Complexity Group at TU Wien, Vienna, Austria.