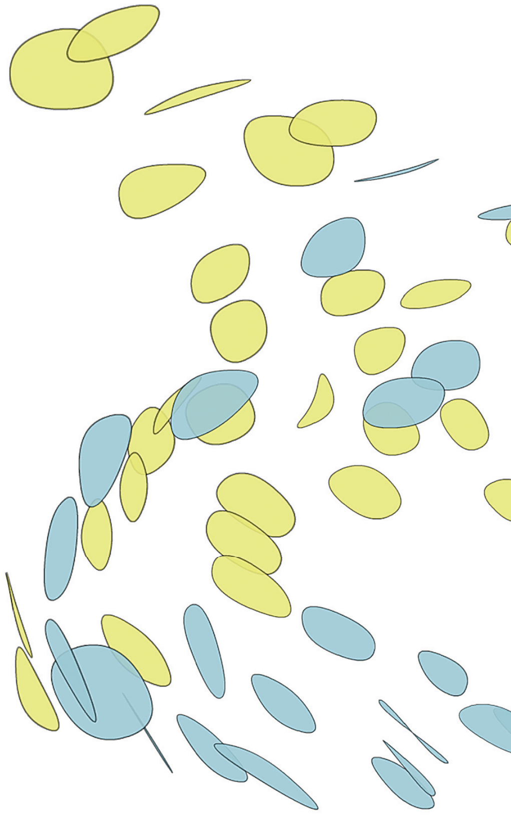


DOI:10.1145/1536616.1536637

Satisfiability solvers can now be effectively deployed in practical applications.

BY SHARAD MALIK AND LINTAO ZHANG

Boolean Satisfiability From Theoretical Hardness to Practical Success

THERE ARE MANY practical situations where we need to satisfy several potentially conflicting constraints. Simple examples of this abound in daily life, for example, determining a schedule for a series of games that resolves the availability of players and venues, or finding a seating assignment at dinner consistent with various rules the host would like to impose. This also applies to applications in computing, for example, ensuring that a hardware/software system functions correctly with its overall behavior constrained by the behavior of its components and their composition, or finding a plan for a robot to reach a goal that is consistent with the moves it can make at any step. While the applications may seem varied, at the core they all have variables whose values we need to determine (for example, the person sitting at a given seat at dinner) and constraints that these variables must satisfy (for example, the host's seating rules).

In its simplest form, the variables are *Boolean* valued (true/false, often represented using 1/0) and *propositional logic formulas* can be used to express the constraints on the variables.[15] In propositional logic the *operators* AND, OR, and NOT (represented by the symbols $\wedge$, $\vee$, and $\neg$ respectively) are used to construct formulas with variables. If x is a Boolean variable and f, f_1 and f_2 are propositional logic formulas (subsequently referred to simply as formulas), then the following recursive definition describes how complex formulas are constructed and evaluated using the constants 0 and 1, the variables, and these operators.

- x is a formula that evaluates to 1 when x is 1, and evaluates to 0 when x is 0
- $\neg f$ is a formula that evaluates to 1 when f evaluates to 0, and 0 when f evaluates to 1
- $f_1 \wedge f_2$ is a formula that evaluates to 1 when f_1 and f_2 both evaluate to 1, and

evaluates to 0 if either f_1 or f_2 evaluate to 0

- $f_1 \vee f_2$ is a formula that evaluates to 0 when f_1 and f_2 both evaluate to 0, and evaluates to 1 if either f_1 or f_2 evaluate to 1

$(x_1 \vee \neg x_2) \wedge x_3$ is an example formula constructed using these rules. Given a valuation of the variables, these rules can be used to determine the valuation of the formula. For example: when $(x_1 = 0, x_2 = 0, x_3 = 1)$, this formula evaluates to 1 and when $(x_3 = 0)$, this formula evaluates to 0, regardless of the values of x_1 and x_2. This example also illustrates how the operators in the formula provide constraints on the variables. In this example, for this formula to be true (evaluate to 1), x_3 must be 1.

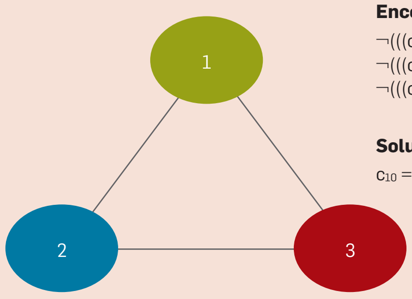
ILLUSTRATION BY GWEN VANHEE

Boolean Satisfiability

A satisfying assignment for a formula is an assignment of the variables such that the formula evaluates to 1. It simultaneously satisfies the constraints imposed by all the operators in the formula. Such an assignment may not always exist. For example the formula $(\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ cannot be satisfied by any of the four possible assignments 0/0, 0/1, 1/0, 1/1 to x_1 and x_2. In this case the problem is overconstrained. This leads us to a definition of the Boolean Satisfiability problem (also referred to as Propositional Satisfiability or just Satisfiability, and abbreviated as SAT): *Given a formula, find a satisfying assignment or prove that none exists.* This is the constructive version of the problem, and one used in practice. A simpler decision version, often used on the theoretical side, just needs to determine if there exists a satisfying assignment for the formula (a yes/no answer). It is easy to see that a solver for the decision version of the problem can easily be used to construct a solution to the constructive version, by solving a series of n decision problems where n is the number of variables in a formula.

Many constraint satisfaction problems dealing with non-Boolean variables can be relatively easily translated to SAT. For example, consider an instance of the classic graph coloring problem where an n-vertex graph needs to be checked for 4-colorability, that is, determining whether each vertex can be colored using one of four possible colors such that no two adjacent vertices have the same color. In this case, the variables are the colors $\{c_0, c_1, c_2, \ldots, c_{n-1}\}$ for the n vertices, and the constraints are that adjacent vertices must have different colors. For this problem the variables are not Boolean and the constraints are not directly expressed with the operators $\{\wedge, \vee, \neg\}$. However, the variables and constraints can be encoded into a propositional formula as follows. Two Boolean variables, c_{i0}, c_{i1}, are used in a two-bit encoding of the four possible values of the color for vertex i. Let i and j be adjacent vertices.

Figure 1. Encoding of graph coloring.



Encoding

$\neg(((c_{10} \wedge c_{20}) \vee (\neg c_{10} \wedge \neg c_{20})) \wedge ((c_{11} \wedge c_{21}) \vee (\neg c_{11} \wedge \neg c_{21}))) \wedge$
$\neg(((c_{10} \wedge c_{30}) \vee (\neg c_{10} \wedge \neg c_{30})) \wedge ((c_{11} \wedge c_{31}) \vee (\neg c_{11} \wedge \neg c_{31}))) \wedge$
$\neg(((c_{30} \wedge c_{20}) \vee (\neg c_{30} \wedge \neg c_{20})) \wedge ((c_{31} \wedge c_{21}) \vee (\neg c_{31} \wedge \neg c_{21})))$

Solution

$c_{10} = 0 \wedge c_{11} = 0 \wedge c_{20} = 0 \wedge c_{21} = 1 \wedge c_{30} = 1 \wedge c_{31} = 0$

The constraint $c_i \neq c_j$ is then expressed as $\neg((c_{i0} == c_{j0}) \wedge (c_{i1} == c_{j1}))$, here $==$ represents equality and thus this condition checks that both bits in the encoding do not have the same value for i and j. Further, $(c_{i0} == c_{j0})$ can be expressed as $(c_{i0} \wedge c_{j0}) \vee (\neg c_{i0} \wedge \neg c_{j0})$, that is, they are both 1 or both 0. Similarly for $(c_{i1} == c_{j1})$. If we take the conjunction of the constraints on each edge, then the resulting formula is satisfiable if and only if the original graph coloring problem has a solution. Figure 1 illustrates an instance of the encoding of the graph coloring problem into a Boolean formula and its satisfying solution.

Encodings have been useful in translating problems from a wide range of domains to SAT, for example, scheduling basketball games,[40] planning in artificial intelligence,[20] validating software models,[17] routing field programmable gate arrays,[28] and synthesizing consistent network configurations.[29] This makes SAT solvers powerful engines for solving constraint satisfaction problems. However, SAT solvers are not always the best engines—there are many cases where specialized techniques work better for various constraint problems, including graph coloring (for example, Johnson et al.[19]). Nonetheless, it is often much easier and more efficient to use off-the-shelf SAT solvers than developing specialized tools from scratch.

One of the more prominent practical applications of SAT has been in the design and verification of digital circuits. Here, the translation to a formula is very straightforward. The functionality of digital circuits can be expressed as compositions of basic logic gates. A logic gate has Boolean input signals and produces Boolean output signals. The output of a gate can be used as an input to another gate. The functions of the basic logic gates are in direct correspondence to the operators $\{\wedge, \vee, \neg\}$. Thus various properties regarding the functionality of logic circuits can be easily translated to formulas. For example, checking that the values of two signals s_1 and s_2 in the logic circuit are always the same is equivalent to checking that their corresponding formulas f_1 and f_2 never differ, that is, $(f_1 \wedge \neg f_2) \vee (\neg f_1 \wedge f_2)$ is not satisfiable.

This technique can be extended to handle more complex properties involving values on sequences of signals, for example, a request is eventually acknowledged. For such problems, techniques that deal with temporal properties of the system, such as model checking, are used.[6] Modern SAT solvers have also been successfully applied for such tasks.[3,26] One of the main difficulties of applying SAT in checking such properties is to find a way to express the concept of "*eventually*." In theory, there is no tractable way to express this using propositional logic. However, in practice it is often good enough to just set a *bound* on the number of steps. For example, instead of asking whether a response to a request will eventually occur, we ask whether there will be a response within k clock cycles, where k is a small fixed number. Similar techniques have also been used in AI planning,[20] for example, instead of determining if a goal is reachable, we ask whether we can reach the goal in k steps. This unrolling technique has been widely adopted in practice, since we often only care about the behavior of the system within a small bounded number of steps.

Theoretical Hardness: SAT and NP-Completeness

The decision version of SAT, that is, determining if a given formula has a satisfying solution, belongs to the class of problems known as *NP-complete*.[8,12] An instance of any one of these problems can be relatively easily transformed into an instance of another. For example, both graph coloring and SAT are NP-complete, and earlier we described how to transform a graph coloring instance to a SAT instance.

All currently known solutions for NP-Complete problems, in the worst case, require runtime that grows exponentially with the size of the instance. Whether there exist subexponential solutions to NP-Complete problems is arguably the most famous open question in computer science.[a] Although there is no definitive conclusion, the answer is widely believed to be in the negative. This exponential growth in time complexity indicates the difficulty of scaling solutions to larger instances.

However, an important part of this characterization is "worst case." This holds out some hope for the "typical case," and more importantly the typical case that might arise in specific problem domains. *In fact, it is exactly the non-adversarial nature of practical instances that is exploited by SAT solvers.*

Solving SAT

Most SAT solvers work with a restricted representation of formulas in conjunctive normal form (CNF), defined as follows. A literal l is either a positive or a negative occurrence of a variable (for example, x or $\neg x$). A clause, c, is the OR of a set of literals, such as $(l_1 \vee l_2 \vee l_3 \ldots \vee l_n)$. A CNF formula is the AND of a set of clauses, such as $(c_1 \wedge c_2 \wedge c_3 \wedge\ c_m)$. An example CNF formula is:

$$(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_4)$$

The restriction to CNF is an active choice made by SAT solvers as it enables their underlying algorithms. Further, this is not a limitation in terms of the formulas that can be handled. Indeed, with the addition of new auxiliary variables; it is easy to translate any formula into CNF with only a linear increase in size.[36] However, this representation is not used exclusively and there has been recent success with

a http://www.claymath.org/millennium/.

solvers for non-clausal representations (for example, NFLSAT[18]).

Most practically successful SAT solvers are based on an approach called systematic search. Figure 2 depicts the search space of a formula. The search space is a tree with each vertex representing a variable and the out edges representing the two decision choices for this variable. For a formula with n variables, there are 2^n leaves in the tree. Each path from the root to a leaf corresponds to a possible assignment to the n variables. The formula may evaluate to 1 or 0 at a leaf (colored green and red respectively). Systematic search, as the name implies, systematically searches the tree and tries to find a green leaf or prove that none exists.

The NP-completeness of the problem indicates that we will likely need to visit an exponential number of vertices in the worst case. The only hope for a practical solver is that by being smart in the search, almost all of the tree can be pruned away and only a minuscule fraction is actually visited in most cases. For an instance with a million variables, which is considered within the reach of modern solvers, the tree has 2^{10^6} leaves, and in reasonable computation time (about a day), we may be able to visit a billion (about 2^{30}) vertices as part of the search—a numerically insignificant fraction of the tree size!

Most search-based SAT solvers are based on the so called *DPLL approach* proposed by Davis, Logemann, and Loveland in a seminal *Communications* paper published in 1962.[9] (This research builds on the work by Davis and Putnam[10] and thus Putnam is often given shared credit for it.). Given a CNF formula, the DPLL algorithm first heuristically chooses an unassigned variable and assigns it a value: either 1 or 0. This is called branching or the *decision* step. The solver then tries to deduce the consequences of the variable assignment using deduction rules. The most widely used deduction rule is the *unit-clause* rule, which states that if a clause in the formula has all but one of its literals assigned 0 and the remaining one is unassigned, then the only way for the clause to evaluate to true, and thus the formula to evaluate to true, is for this last unassigned literal to be assigned to 1. Such clauses are called unit clauses and the forced assignments are called implications. This rule is applied iteratively until no unit clause exists. Note that this deduction is enabled by the CNF representation and is the main reason for SAT solvers preferring this form.

If at some point there is a clause in the formula with all of its literals evaluating to 0, then the formula cannot be true under the current assignment. This is called a *conflict* and this clause is referred to as a conflicting clause. A conflict indicates that some of the earlier decision choices cannot lead to a satisfying solution and the solver has to backtrack and try a different branch value. It accomplishes this by finding the most recent decision variable for which both branches have not been taken, flip its value, undo all variable assignments after that decision, and run the deduction process again. Otherwise, if no such conflicting clause exists, the solver continues by branching on another unassigned variable. The search stops either when all variables are assigned a value, in which case we have hit a green leaf and the formula is satisfiable, or when a conflicting clause exists when all branches have been explored, in which case the formula is unsatisfiable.

Consider the application of the algorithm to the formula shown in Figure 2. At the beginning the solver branches on variable x_1 with value 1. After branching, the first clause becomes unit and the remaining free literal $\neg x_2$ is implied to 1, which means x_2 must be 0. Now the second clause becomes unit and $\neg x_3$ is implied to 1. Then $\neg x_4$ is implied to 1 due to the third clause. At this point the formula is satisfied, and the satisfying assignment corresponds to the 8th leaf node from the left in the search tree. (This path is marked in bold in the figure.) As we can see, by applying the unit-clause rule, a single branching leads directly to the satisfying solution.

Many significant improvements in the basic DPLL algorithm have been

Figure 2. Search space of a formula.

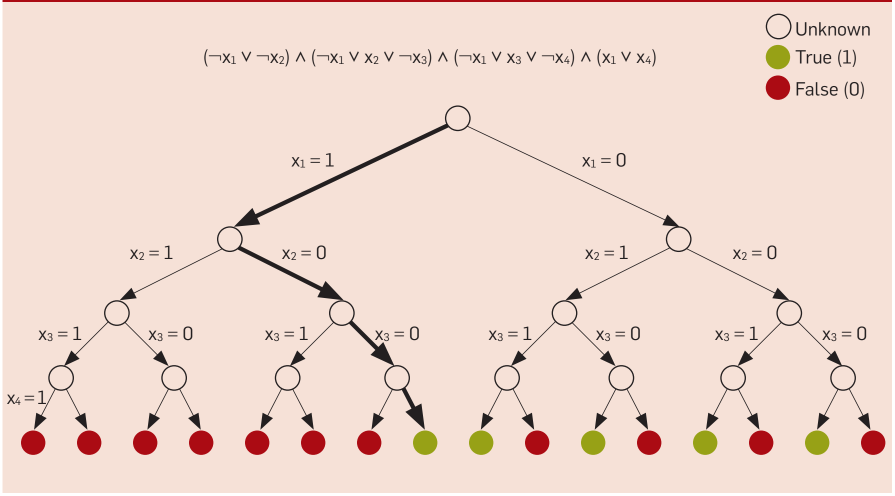
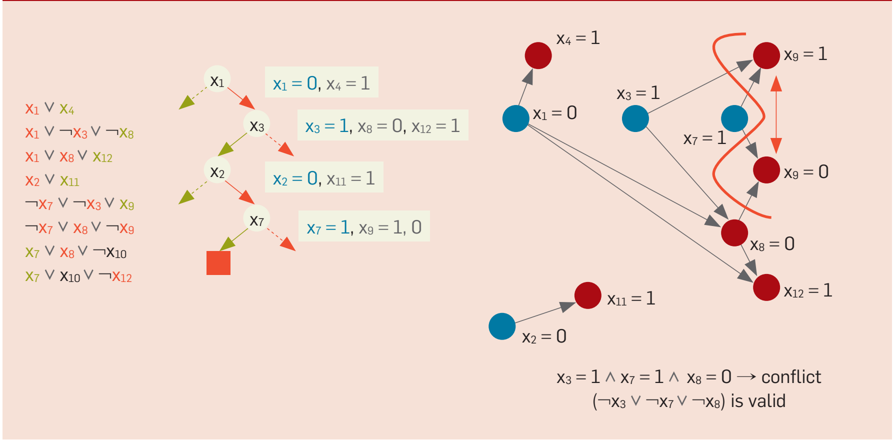


Figure 3. Conflict-driven learning and non-chronological backtracking.



proposed over the years. In particular, a technique called conflict-driven learning and non-chronological backtracking[2, 24] has greatly enhanced the power of DPLL SAT solvers on problem instances arising from real applications, and has become a key element of modern SAT solvers. The technique is illustrated in Figure 3. The column on the left lists the clauses in the example formula. The colors of the literals show the current assignments during the search (red representing 0, green 1, and black representing unassigned). The middle graph shows the branching and implications at the current point in the search. At each vertex the branching assignment is shown in blue and the implications in gray. The first branching is on x_1, and it implies x_4=1 (because of the first clause), the second branching is on x_3, and it implies x_8=0 and x_{12}=1 and so on. The right graph shows the implication relationships between variables. For example, x_4=1 is implied because of x_1=0, so there is a directed edge from node x_1=0 to node x_4=1. x_8=0 is implied because of both x_1=0 and x_3=1 (the red literals in the second clause), therefore, these nodes have edges leading to x_8=0.

After branching on x_7 and implying x_9=1 because of the 5th clause, we find that the 6th clause becomes a conflicting clause and the solver has to backtrack. Instead of flipping the last decision variable x_7 and trying x_7=0, we can learn some information from the conflict. From the implication graph, we see that there is a conflict because x_9 is implied to be both 1 and 0. If we consider a cut (shown as the orange line) separating the conflicting implications from the branching decisions, we know that once the assignments corresponding to the cut edges are made, we will end up with a conflict, since no further decisions are made. Thus, the edges that cross the cut are, in some sense, responsible for the conflict. In the example, x_3, x_7, and x_8 have edges cross the cut, thus the combination of x_3=1, x_7=1, and x_8=0 results in the conflict. We can learn from this and ensure that this assignment combination is not tried in the future. This is accomplished by recording the condition ($\neg x_3 \vee \neg x_7 \vee x_8$). This clause, referred to as a learned clause, can be added to the formula. While it is redundant in the sense that it is implied by the formula, it is nonetheless useful as it prevents search from ever making the assignment (x_3=1, x_7=1, x_8=0) again.

Further, because of this learned clause, $x_7 = 1$ is now implied after the second decision, and we can backtrack to this earlier decision level as the choice of $x_2 = 0$ is irrelevant to the current conflict. Since such backtracking skips branches, it is called non-chronological backtracking and helps prune away unsatisfiable parts of the search space.

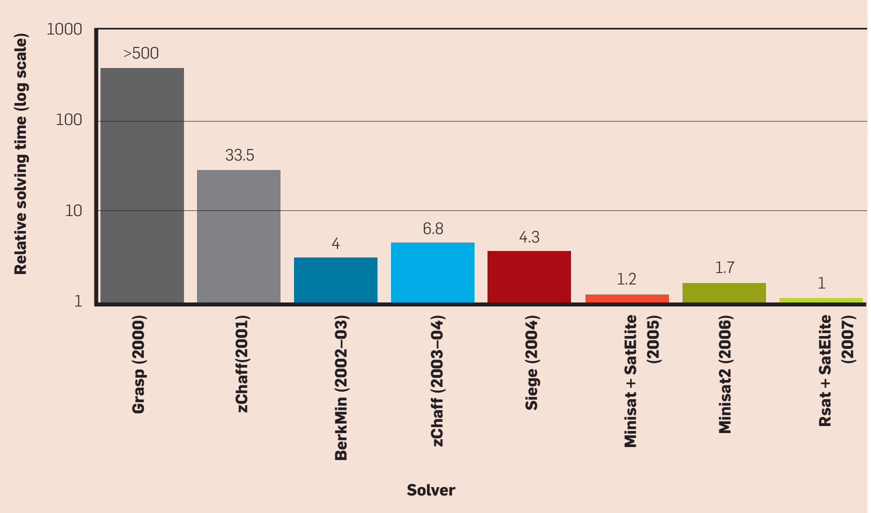
Recent Results

Recent work has exposed several significant areas of improvement now integral to modern SAT solvers.[22] The first deals with efficient implementation of the unit-clause rule using a technique called two-literal watching. The second area relates to improvements in the branching step by focusing on exhausting local sub-spaces before moving to new spaces. This is accomplished by placing increased emphasis on variables present in recently added conflict clauses. Another commonly used technique is random restart,[13] which periodically restarts the search while retaining the learned clauses from the current search to avoid being stuck in a search sub-space for too long. Other recent directions include formula preprocessing for clause and variable elimination,[11] considering algorithm portfolios that use empirical hardness models to choose among their constituent solvers on a per-instance basis[39] and using learning techniques to adjust parameters of heuristics.[16] With the advent of multicore processing, there is emerging interest in efficient multi-core implementations of parallel SAT solvers.[14]

The original Davis Putnam algorithm[10] based on resolution is often regarded as the first algorithm for SAT and has great theoretical and historical significance. However, this algorithm suffers from a space growth problem that makes it impractical. Reduced Ordered Binary Decision Diagrams (ROBDDs)[5] are a canonical representation of logic functions, that is, each function has a unique representation for a fixed variable ordering. Thus, ROBDDS can be used directly for SAT. However, ROBDDs also face space limitations with increasing instance size. Stålmarck's algorithm[35] uses breadth-first search instead of depth-first search as in DPLL, and has been shown to be practically useful. Its performance relative to DPLL based solvers is unclear as public versions of efficient implementations of Stålmarck's algorithm are not available due to its proprietary nature.

When represented in CNF, SAT can be regarded as a discrete optimization problem with the objective to maximize the number of satisfied clauses. If this max value is equal to the total number of clauses, then the instance is satisfiable. Many discrete optimization techniques have been explored in the SAT context, including simulated annealing,[33] tabu search,[25] neural networks,[34] and genetic algorithms.[23]

Figure 4. Speedup of SAT solvers in recent years.



A variation of the optimization approach, first proposed in the early 1990s, solves SAT using local search (for example, GSAT[31]). The algorithm first randomly selects a value for each variable, and calculates how many clauses are satisfied. If not all clauses are satisfied, it repeatedly flips the value of a variable to increase the number of the clauses satisfied. If no such variable is available, it accepts a decrease in the objective function by either flipping a random variable, or restarting from a fresh set of variable assignments. This is accelerated further, by confining the flips to literals in clauses not satisfied by the current assignment.[30] This simple algorithm, when carefully implemented, is surprisingly effective on certain classes of SAT instances. Unfortunately, this algorithm is incomplete in the sense that while it may be able to find an assignment for a satisfiable SAT instance, it cannot prove an instance to be unsatisfiable. More recently, incomplete solvers based on a technique called survey propagation[4] have been found to be very effective for certain classes of SAT instances and have attracted much attention in the theory community.

The Role of Benchmarks

It is important to note the role of practical benchmarks in the development of modern SAT solvers. These benchmarks are critical in tuning the solvers to various classes of practical instances (that is, instances generated from real-world applications). While we do not have deep insight into how these solvers exploit the special structure found in these instances, we do know that the structure is critical in our ability to tackle them. (There exists some recent work that provides initial insights into the effect of structure on DPLL search.[37,38]) Experimental research in SAT solvers has been enabled in large part by benchmarks put forward collectively by the research community, and the challenge in the form of a SAT solver competition that is held regularly with the International Conference on Theory and Applications of Satisfiability Testing (SAT).[b] The community has also benefited from the SATLive portal, which has provided widespread dissemination of links to SAT articles and software.[c]

b http://www.satcompetition.org/.

One of the more prominent practical applications of SAT has been in the design and verification of digital circuits. The functionality of digital circuits can be expressed as compositions of basic logic gates.

Figure 4 provides some data on the improvements in SAT solvers at the SAT Competition in recent years.[d] It plots the relative solving times for a set of solvers developed over the last 10 years. This includes solvers that placed first in the industrial benchmarks category of the SAT competitions. The solvers were run on a set of benchmarks from hardware and software verification (not used in the competitions).[32] This is normalized to the best solver in the 2007 competition (RSAT with the SatElite preprocessor). The slow-down of the Grasp solver is a lower bound, since it could not complete some of the benchmarks in the 10,000-second time limit. While this study is limited to a specific set of benchmarks, it is indicative of the progress in SAT solvers since 2000.

Industrial Impact

SAT solvers are maturing to the point that developers are using them in a range of application domains, much like mathematical programming tools or linear equation solvers. Early use of SAT was seen in planning in artificial intelligence with practical use in space exploration.[27] Recent increases in the capacity of commercial solvers has enabled widespread use in the electronic design automation (EDA) industry as the reasoning engine behind verification and testing tools such as automatic test pattern generators,[21] equivalence checkers, and property checkers. SAT-based bounded model checkers have been used in industrial microprocessor verification.[7] More recently, SAT has also been used in tools for software verification and debugging, for example, industrial verification of device drivers using SAT-based model checking,[e] as well as SAT-based static analysis.[f] Outside of verification and testing, SAT techniques have also been applied in configuration management such as resolving software package dependencies.[g]

c http://www.satlive.org/.
d Provided by Sanjit Seshia, UC Berkeley.
e http://www.microsoft.com/whdc/DevTools/tools/SDV.mspx.
f http://www.coverity.com/index.html.
g http://news.opensuse.org/2008/06/06/sneak-peeks-at-opensuse-110-package-management-with-duncan-mac-vicar/.

Beyond SAT

The success with SAT solvers has emboldened researchers to consider problems related to, but more difficult than SAT. The most promising of these is Satisfiability Modulo Theories (SMT) that has received significant attention in recent years.

In SAT, the variables are assumed to be constrained only by the clauses in the formula. SMT extends SAT by considering the case when the variables may be connected by one or more underlying theories. For example, consider the formula $(x_1 \wedge \neg x_2 \wedge x_3)$. This formula is clearly satisfiable with $(x_1 = 1, x_2=0, x_3=1)$. However, if x_1, x_2 and x_3 represent the following relationships among the real variables y_1 and y_2:

x_1: $y_1 < 0$
x_2: $y_1 + y_2 < 1$
x_3: $y_2 < 0$

Then, in fact, there is no assignment to y_1 and y_2 for which $(x_1 = 1, x_2=0, x_3=1)$, i.e., y_1 and y_2 cannot be both negative and their sum at least one. Thus, the original formula is unsatisfiable given this underlying relationship. In this example, the specific theory used to determine the validity of a satisfying assignment is Linear Real Arithmetic. Emerging SMT solvers can incorporate reasoning for a range of theories such as Linear Integer Arithmetic, Difference Logic, Arrays, Lists, Uninterpreted Functions and many others, including their combinations.[1] The theoretical difficulty depends on the specific theories considered. SMT is seeing rapid progress and initial commercial use in software verification.

Conclusion

The success with SAT has led to its widespread commercial use in certain domains such as design and verification of hardware and software systems. There is even a sense in parts of the computer science community that this problem has been successfully tamed in practice. This is probably too optimistic a view. There are still enough instances that are difficult for current solvers, and it is unclear if they will be able to handle the change in scale/nature of instances from yet unseen domains. However, there is definitely a sense of confidence that we will be able to continue to strengthen our solvers.

Given its theoretical hardness, the practical success of SAT has come as a surprise to many in the computer science community. The combination of strong practical drivers and open competition in this experimental research effort created enough momentum to overcome the pessimism based on theory. Can we take these lessons to other problems and domains? C

References

1. Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. Satisfiability modulo theories. A. Biere, H. van Maaren, T. Walsh, Eds. *Handbook of Satisfiability 4*, 8 (2009), IOS Press, Amsterdam.
2. Bayardo R., and Schrag, R. Using CSP look-back techniques to solve real-world SAT instances. *National Conference on Artificial Intelligence*, 1997.
3. Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. Symbolic model checking without BDDs. *Tools and Algorithms for the Analysis and Construction of Systems*, 1999.
4. Braunstein, A., Mezard, M., and Zecchina, R. Survey propagation: An algorithm for Satisfiability. *Random Structures and Algorithms 27* (2005), 201–226.
5. Bryant, R.E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35* (1986), 677–691.
6. Clarke, E.M., Grumberg, O., and Peled, D.A. *Model Checking.* MIT Press, Cambridge, MA, 1999.
7. Copty, F., Fix, L., Fraer, R., Giunchiglia, E., Kamhi, G., Tacchella, A., and Vardi, M.Y. Benefits of bounded model checking at an industrial setting. *Proceedings of the 13th International Conference on Computer-Aided Verification*, 2001.
8. Cook, S.A. The complexity of theorem-proving procedures. *Third Annual ACM Symposium on Theory of Computing*, 1971.
9. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. *Comm. ACM 5* (1962), 394–397.
10. Davis, M., and Putnam, H. A computing procedure for quantification theory. *JACM 7* (1960), 201–215.
11. Eén, N., and Biere, A. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2005.
12. Garey, M.R., and Johnson, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979
13. Gomes, C. P., Selman, B., and Kautz, H. Boosting combinatorial search through randomization. In *Proceedings of National Conference on Artificial Intelligence* (Madison, WI, 1998).
14. Hamadi, Y., Jabbour, S., and Sais, L. ManySat: Solver description. Microsoft Research, TR-2008-83.
15. Huth, M. and Ryan, M. *Logic in Computer Science: Modeling and Reasoning about Systems.* Cambridge University Press, 2004.
16. Hutter, F., Babic, D., Hoos, H.H., and Hu, A. J. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, (Austin, TX, Nov. 2007).
17. Jackson, D., and Vaziri, M., Finding bugs with a constraint solver. In *Proceedings of the International Symposium on Software Testing and Analysis* (Portland, OR, 2000).
18. Jain, H. Verification using satisfiability checking, predicate abstraction, and Craig interpolation. Ph.D. Thesis, Carnegie-Mellon University, School of Computer Science, CMU-CS-08-146, 2008.
19. Johnson, D.S., Mehrotra, A., and Trick, M. A. Preface: Special issue on computational methods for graph coloring and its generalizations. *Discrete Applied Mathematics 156*, 2; Computational Methods for Graph Coloring and its Generalizations. (Jan. 15, 2008), 145–146.
20. Kautz, H. and Selman, B. Planning as satisfiability. *European Conference on Artificial Intelligence*, 1992.
21. Larrabee, T. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design* (Jan. 1992) 4–15.
22. Madigan, M.W., Madigan, C.F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Conference on Design Automation.* (New York, NY, 2001).
23. Marchiori, E. and Rossi, C. A flipping genetic algorithm for hard 3-SAT problems. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Orlando, FL, 1999), 393–400.
24. Marques-Silva, J.P, and Sakallah, K.A. Conflict analysis in search algorithms for propositional satisfiability. *IEEE International Conference on Tools with Artificial Intelligence*, 1996.
25. Mazure, B., Sas L., and Grgoire, E., Tabu search for SAT. In *Proceedings of the 14th National Conference on Artificial Intelligence* (Providence, RI, 1997).
26. McMillan, K.L., Applying SAT methods in unbounded symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*. Lecture Notes In Computer Science 2404 (2002). Springer-Verlag, London, 250–264.
27. Muscettola, N., Pandurang Nayak, P., Pell, B., and Williams, B.C. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence 103*, 1–2, (1998), 5–47.
28. Nam, G.-J., Sakallah, K. A., and Rutenbar, R.A. Satisfiability-based layout revisited: Detailed routing of complex FPGAs via search-based Boolean SAT. *International Symposium on Field-Programmable Gate Arrays* (Monterey, CA, 1999).
29. Narain, S., Levin, G., Kaul, V., Malik, S. Declarative infrastructure configuration and debugging. *Journal of Network Systems and Management, Special Issue on Security Configuration.* Springer, 2008.
30. Selman, B., Kautz, H.A., and Cohen, B. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence* (Seattle, WA, 1994). American Association for Artificial Intelligence, Menlo Park, CA, 337–343.
31. Selman, B., Levesque, H., and Mitchell, D. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, (1992) 440–446.
32. Seshia, S.A., Lahiri, S.K., and Bryant, R.E. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proceedings of the 40th Conference on Design Automation* (Anaheim, CA, June 2–6, 2003). ACM, NY, 425–430; http://doi.acm.org/10.1145/775832.775945.
33. Spears, W.M. Simulated annealing for hard satisfiability problems. *Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenge. DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* D.S. Johnson and M.A. Trick, Eds. American Mathematical Society (1993), 533–558.
34. Spears, W. M. A NN algorithm for Boolean satisfiability problems. In *Proceedings of the 1996 International Conference on Neural Networks*, 1121–1126.
35. Stålmarck, G. A system for determining prepositional logic theorems by applying values and rules to triplets that are generated from a formula. U.S. Patent Number 5276897, 1994.
36. Tseitin, G. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2* (1968), 115–125. Reprinted in *Automation of reasoning vol. 2.* J. Siekmann and G. Wrightson, Eds. Springer Verlag, Berlin, 1983, 466–483.
37. Williams, R., Gomes, C., and Selman, B. Backdoors to typical case complexity. In *Proceedings. of the 18th International Joint Conference on Artificial Intelligence* (2003), 1173–1178.
38. Williams, R., Gomes, C., and Selman, B. On the connections between heavy-tails, backdoors, and restarts in combinatorial search. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2003.
39. Xu, L., Hutter, F., Hoos, H. H., Leyton-Brown, K. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research 32*, (2008), 565–606.
40. Zhang, H. Generating college conference basketball schedules by a SAT solver. In *Proceedings of the 5th International Symposium on Theory and Applications of Satisfiability Testing.* (Cincinnati, OH, 2002).

Sharad Malik (sharad@princeton.edu) is a professor in the Department of Electrical Engineering at Princeton University, Princeton, NJ.

Lintao Zhang (lintaoz@microsoft.com) is a researcher at Microsoft Research Asia, Beijing, China.