SAT solver

In computer science and formal methods, a **SAT solver** is a computer program which aims to solve the Boolean satisfiability problem. On input a formula over Boolean variables, such as "(*x* or *y*) and (*x* or not *y*)", a SAT solver outputs whether the formula is satisfiable, meaning that there are possible values of *x* and *y* which make the formula true, or unsatisfiable, meaning that there are no such values of *x* and *y*. In this case, the formula is satisfiable when *x* is true, so the solver should return "satisfiable". Since the introduction of algorithms for SAT in the 1960s, modern SAT solvers have grown into complex software artifacts involving a large number of heuristics and program optimizations to work efficiently.

By a result known as the Cook–Levin theorem, Boolean satisfiability is an NP-complete problem in general. As a result, only algorithms with exponential worst-case complexity are known. In spite of this, efficient and scalable algorithms for SAT were developed during the 2000s, which have contributed to dramatic advances in the ability to automatically solve problem instances involving tens of thousands of variables and millions of constraints.[1]

SAT solvers often begin by converting a formula to conjunctive normal form. They are often based on core algorithms such as the DPLL algorithm, but incorporate a number of extensions and features. Most SAT solvers include time-outs, so they will terminate in reasonable time even if they cannot find a solution, with an output such as "unknown" in the latter case. Often, SAT solvers do not just provide an answer, but can provide further information including an example assignment (values for *x*, *y*, etc.) in case the formula is satisfiable or minimal set of unsatisfiable clauses if the formula is unsatisfiable.

Modern SAT solvers have had a significant impact on fields including software verification, program analysis, constraint solving, artificial intelligence, electronic design automation, and operations research. Powerful solvers are readily available as free and open-source software and are built into some programming languages such as exposing SAT solvers as constraints in constraint logic programming.

Overview

A *Boolean formula* is any expression that can be written using Boolean (propositional) variables *x*, *y*, *z*, ... and the Boolean operations AND, OR, and NOT. For example,

(*x* AND *y*) OR (*x* AND (NOT *z*))

An *assignment* consists of choosing, for each variable, an assignment TRUE or FALSE. For any assignment *v*, the Boolean formula can be evaluated, and evaluates to true or false. The formula is *satisfiable* if there exists an assignment (called a *satisfying assignment*) for which the formula evaluates to true.

The *Boolean satisfiability problem* is the decision problem which asks, on input a Boolean formula, to determine whether the formula is satisfiable or not. This problem is NP-complete.

Core algorithms

SAT solvers are usually developed using one of two core approaches: the Davis–Putnam–Logemann–Loveland algorithm (DPLL) and conflict-driven clause learning (CDCL).

DPLL

A DPLL SAT solver employs a systematic backtracking search procedure to explore the (exponentially sized) space of variable assignments looking for satisfying assignments. The basic search procedure was proposed in two seminal papers in the early 1960s (see references below) and is now commonly referred to as the DPLL algorithm.[2][3] Many modern approaches to practical SAT solving are derived from the DPLL algorithm and share the same structure. Often they only improve the efficiency of certain classes of SAT problems such as instances that appear in industrial applications or randomly generated instances.[4] Theoretically, exponential lower bounds have been proved for the DPLL family of algorithms.

CDCL

Modern SAT solvers (developed in the 2000s) come in two flavors: "conflict-driven" and "look-ahead". Both approaches descend from DPLL.[4] Conflict-driven solvers, such as conflict-driven clause learning (CDCL), augment the basic DPLL search algorithm with efficient conflict analysis, clause learning, backjumping, a "two-watched-literals" form of unit propagation, adaptive branching, and random restarts. These "extras" to the basic systematic search have been empirically shown to be essential for handling the large SAT instances that arise in electronic design automation (EDA).[5] Most state-of-the-art SAT solvers are based on the CDCL framework as of 2019.[6] Well known implementations include Chaff[7] and GRASP.[8]

Look-ahead solvers have especially strengthened reductions (going beyond unit-clause propagation) and the heuristics, and they are generally stronger than conflict-driven solvers on hard instances (while conflict-driven solvers can be much better on large instances which actually have an easy instance inside).

The conflict-driven MiniSAT, which was relatively successful at the 2005 SAT competition, only has about 600 lines of code. A modern Parallel SAT solver is ManySAT.[9] It can achieve super linear speed-ups on important classes of problems. An example for look-ahead solvers is march_dl, which won a prize at the 2007 SAT competition. Google's CP-SAT solver, part of OR-Tools, won gold medals at the Minizinc constraint programming competitions in 2018, 2019, 2020, and 2021.

Certain types of large random satisfiable instances of SAT can be solved by survey propagation (SP). Particularly in hardware design and verification applications, satisfiability and other logical properties of a given propositional formula are sometimes decided based on a representation of the formula as a binary decision diagram (BDD).

Different SAT solvers will find different instances easy or hard, and some excel at proving unsatisfiability, and others at finding solutions. All of these behaviors can be seen in the SAT solving contests.[10]

Parallel approaches

Parallel SAT solvers come in three categories: portfolio, divide-and-conquer and parallel local search algorithms. With parallel portfolios, multiple different SAT solvers run concurrently. Each of them solves a copy of the SAT instance, whereas divide-and-conquer algorithms divide the problem between the processors. Different approaches exist to parallelize local search algorithms.

The International SAT Solver Competition has a parallel track reflecting recent advances in parallel SAT solving. In 2016,[11] 2017[12] and 2018,[13] the benchmarks were run on a shared-memory system with 24 processing cores, therefore solvers intended for distributed memory or manycore processors might have fallen short.

Portfolios

In general there is no SAT solver that performs better than all other solvers on all SAT problems. An algorithm might perform well for problem instances others struggle with, but will do worse with other instances. Furthermore, given a SAT instance, there is no reliable way to predict which algorithm will solve this instance particularly fast. These limitations motivate the parallel portfolio approach. A portfolio is a set of different algorithms or different configurations of the same algorithm. All solvers in a parallel portfolio run on different processors to solve of the same problem. If one solver terminates, the portfolio solver reports the problem to be satisfiable or unsatisfiable according to this one solver. All other solvers are terminated. Diversifying portfolios by including a variety of solvers, each performing well on a different set of problems, increases the robustness of the solver.[14]

Many solvers internally use a random number generator. Diversifying their seeds is a simple way to diversify a portfolio. Other diversification strategies involve enabling, disabling or diversifying certain heuristics in the sequential solver.[15]

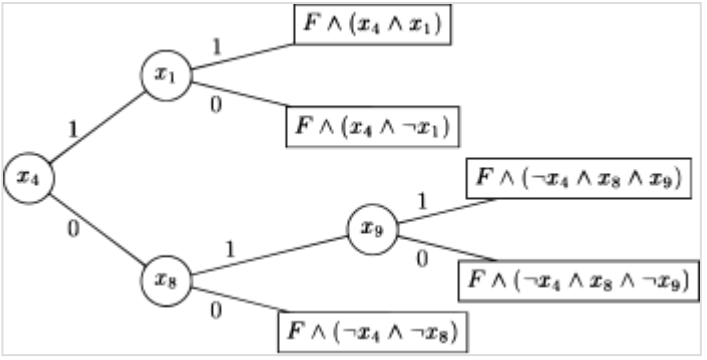
One drawback of parallel portfolios is the amount of duplicate work. If clause learning is used in the sequential solvers, sharing learned clauses between parallel running solvers can reduce duplicate work and increase performance. Yet, even merely running a portfolio of the best solvers in parallel makes a competitive parallel solver. An example of such a solver is PPfolio.[16][17] It was designed to find a lower bound for the performance a parallel SAT solver should be able to deliver. Despite the large amount of duplicate work due to lack of optimizations, it performed well on a shared memory machine. HordeSat[18] is a parallel portfolio solver for large clusters of computing nodes. It uses differently configured instances of the same sequential solver at its core. Particularly for hard SAT instances HordeSat can produce linear speedups and therefore reduce runtime significantly.

In recent years parallel portfolio SAT solvers have dominated the parallel track of the International SAT Solver Competitions. Notable examples of such solvers include Plingeling and painless-mcomsps.[19]

Divide-and-conquer

In contrast to parallel portfolios, parallel divide-and-conquer tries to split the search space between the processing elements. Divide-and-conquer algorithms, such as the sequential DPLL, already apply the technique of splitting the search space, hence their extension towards a parallel algorithm is straight forward. However, due to techniques like unit propagation, following a division, the partial problems may differ significantly in complexity. Thus the DPLL algorithm typically does not process each part of the search space in the same amount of time, yielding a challenging load balancing problem.[14]

Due to non-chronological backtracking, parallelization of conflict-driven clause learning is more difficult. One way to overcome this is the Cube-and-Conquer paradigm.[20] It suggests solving in two phases. In the "cube" phase the Problem is divided into many thousands, up to millions, of sections. This is done by a look-ahead solver, that finds a set of partial configurations called "cubes". A cube can also be seen as a conjunction of a subset of variables of the original formula. In conjunction with the formula, each of the cubes forms a new formula. These formulas can be solved independently and concurrently by conflict-driven solvers. As the disjunction of these formulas is equivalent to the original formula, the problem is reported to be satisfiable, if one of the formulas is satisfiable. The look-ahead solver is favorable for small but hard problems,[21] so it is used to gradually divide the problem into multiple sub-problems. These sub-problems are easier but still large which is the ideal form for a conflict-driven solver. Furthermore, look-ahead solvers consider the entire problem whereas conflict-driven solvers make decisions based on information that is much more local. There are three heuristics involved in the cube phase. The variables in the cubes are chosen by the decision heuristic. The direction heuristic decides which variable assignment (true or false) to explore first. In satisfiable problem instances, choosing a satisfiable branch first is beneficial. The cutoff heuristic decides when to stop expanding a cube and instead forward it to a sequential conflict-driven solver. Preferably the cubes are similarly complex to solve.[20]



Cube phase for the formula F. The decision heuristic chooses which variables (circles) to assign. After the cutoff heuristic decides to stop further branching, the partial problems (rectangles) are solved independently using CDCL.

Treengeling is an example for a parallel solver that applies the Cube-and-Conquer paradigm. Since its introduction in 2012 it has had multiple successes at the International SAT Solver Competition. Cube-and-Conquer was used to solve the Boolean Pythagorean triples problem.[22]

Cube-and-Conquer is a modification or a generalization of the DPLL-based Divide-and-conquer approach used to compute the Van der Waerden numbers w(2;3,17) and w(2;3,18) in 2010 [23] where both the phases (splitting and solving the partial problems) were performed using DPLL.

Local search

One strategy towards a parallel local search algorithm for SAT solving is trying multiple variable flips concurrently on different processing units.[24] Another is to apply the aforementioned portfolio approach, however clause sharing is not possible since local search solvers do not produce clauses. Alternatively, it

is possible to share the configurations that are produced locally. These configurations can be used to guide the production of a new initial configuration when a local solver decides to restart its search.[25]

Randomized approaches

Algorithms that are not part of the DPLL family include stochastic local search algorithms. One example is WalkSAT. Stochastic methods try to find a satisfying interpretation but cannot deduce that a SAT instance is unsatisfiable, as opposed to complete algorithms, such as DPLL.[4]

In contrast, randomized algorithms like the PPSZ algorithm by Paturi, Pudlak, Saks, and Zane set variables in a random order according to some heuristics, for example bounded-width resolution. If the heuristic can't find the correct setting, the variable is assigned randomly. The PPSZ algorithm has a runtime of $O(1.308^n)$ for 3-SAT. This was the best-known runtime for this problem until 2019, when Hansen, Kaplan, Zamir and Zwick published a modification of that algorithm with a runtime of $O(1.307^n)$ for 3-SAT. The latter is currently the fastest known algorithm for k-SAT at all values of k. In the setting with many satisfying assignments the randomized algorithm by Schöning has a better bound.[26][27][28]

Applications

In mathematics

SAT solvers have been used to assist in proving mathematical theorems through computer-assisted proof. In Ramsey theory, several previously unknown Van der Waerden numbers were computed with the help of specialized SAT solvers running on FPGAs.[29][30] In 2016, Marijn Heule, Oliver Kullmann, and Victor Marek solved the Boolean Pythagorean triples problem by using a SAT solver to show that there is no way to color the integers up to 7825 in the required fashion.[31][32] Small values of the Schur numbers were also computed by Heule using SAT solvers.[33]

In software verification

SAT solvers are used in formal verification of hardware and software. In model checking (in particular, bounded model checking), SAT solvers are used to check whether a finite-state system satisfies a specification of its intended behavior.[34][35]

SAT solvers are the core component on which satisfiability modulo theories (SMT) solvers are built, which are used for problems such as job scheduling, symbolic execution, program model checking, program verification based on hoare logic, and other applications.[36] These techniques are also closely related to constraint programming and logic programming.

In other areas

In operations research, SAT solvers have been applied to solve optimization and scheduling problems.[37]

In social choice theory, SAT solvers have been used to prove impossibility theorems.[38] Tang and Lin used SAT solvers to prove Arrow's theorem and other classic impossibility theorems. Geist and Endriss used it to find new impossibilities related to set extensions. Brandt and Geist used this approach to prove an impossibility about strategyproof tournament solutions. Other authors used this technology to prove new impossibilities about the no-show paradox, half-way monotonicity, and probabilistic voting rules. Brandl, Brandt, Peters and Stricker used it to prove the impossibility of a strategyproof, efficient and fair rule for fractional social choice.[39]

See also

- Category:SAT solvers
- Computer-assisted proof
- Satisfiability modulo theories

References

1. Ohrimenko, Olga; Stuckey, Peter J.; Codish, Michael (2007), "Propagation = Lazy Clause Generation", *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, vol. 4741, pp. 544–558, CiteSeerX 10.1.1.70.5471 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.70.5471), doi:10.1007/978-3-540-74970-7_39 (https://doi.org/10.1007%2F978-3-540-74970-7_39), ISBN 978-3-540-74969-1, "modern SAT solvers can often handle problems with millions of constraints and hundreds of thousands of variables"
2. Davis, M.; Putnam, H. (1960). "A Computing Procedure for Quantification Theory". *Journal of the ACM*. **7** (3): 201. doi:10.1145/321033.321034 (https://doi.org/10.1145%2F321033.321034). S2CID 31888376 (https://api.semanticscholar.org/CorpusID:31888376).
3. Davis, M.; Logemann, G.; Loveland, D. (1962). "A machine program for theorem-proving" (http://www.ensiie.fr/~blazy/ipr/article2.pdf) (PDF). *Communications of the ACM*. **5** (7): 394–397. doi:10.1145/368273.368557 (https://doi.org/10.1145%2F368273.368557). hdl:2027/mdp.39015095248095 (https://hdl.handle.net/2027%2Fmdp.39015095248095). S2CID 15866917 (https://api.semanticscholar.org/CorpusID:15866917).
4. Zhang, Lintao; Malik, Sharad (2002), "The Quest for Efficient Boolean Satisfiability Solvers", *Computer Aided Verification*, Lecture Notes in Computer Science, vol. 2404, Springer Berlin Heidelberg, pp. 17–36, doi:10.1007/3-540-45657-0_2 (https://doi.org/10.1007%2F3-540-45657-0_2), ISBN 978-3-540-43997-4
5. Vizel, Y.; Weissenbacher, G.; Malik, S. (2015). "Boolean Satisfiability Solvers and Their Applications in Model Checking". *Proceedings of the IEEE*. **103** (11): 2021–2035. doi:10.1109/JPROC.2015.2455034 (https://doi.org/10.1109%2FJPROC.2015.2455034). S2CID 10190144 (https://api.semanticscholar.org/CorpusID:10190144).
6. Möhle, Sibylle; Biere, Armin (2019). "Backing Backtracking". *Theory and Applications of Satisfiability Testing – SAT 2019* (http://fmv.jku.at/papers/MoehleBiere-SAT19.pdf) (PDF). Lecture Notes in Computer Science. Vol. 11628. pp. 250–266. doi:10.1007/978-3-030-24258-9_18 (https://doi.org/10.1007%2F978-3-030-24258-9_18). ISBN 978-3-030-24257-2. S2CID 195755607 (https://api.semanticscholar.org/CorpusID:195755607).
7. Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; Malik, S. (2001). "Chaff: Engineering an Efficient SAT Solver" (http://www.princeton.edu/~chaff/publication/DAC2001v56.pdf) (PDF). *Proceedings of the 38th conference on Design automation (DAC)*. p. 530. doi:10.1145/378239.379017 (https://doi.org/10.1145%2F378239.379017). ISBN 1581132972. S2CID 9292941 (https://api.semanticscholar.org/CorpusID:9292941).

8. Marques-Silva, J. P.; Sakallah, K. A. (1999). "GRASP: a search algorithm for propositional satisfiability" (https://web.archive.org/web/20161104020512/http://embedded.eecs.berkeley.edu/Alumni/wjiang/ee219b/grasp.pdf) (PDF). *IEEE Transactions on Computers*. **48** (5): 506. doi:10.1109/12.769433 (https://doi.org/10.1109%2F12.769433). Archived from the original (http://embedded.eecs.berkeley.edu/Alumni/wjiang/ee219b/grasp.pdf) (PDF) on 2016-11-04. Retrieved 2015-08-28.
9. http://www.cril.univ-artois.fr/~jabbour/manysat.htm
10. "The international SAT Competitions web page" (http://www.satcompetition.org/). Retrieved 2007-11-15.
11. "SAT Competition 2016" (https://baldur.iti.kit.edu/sat-competition-2016/index.php?cat=tracks). *baldur.iti.kit.edu*. Retrieved 2020-02-13.
12. "SAT Competition 2017" (https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=tracks). *baldur.iti.kit.edu*. Retrieved 2020-02-13.
13. "SAT Competition 2018" (http://sat2018.forsyte.tuwien.ac.at/index.php?cat=tracks). *sat2018.forsyte.tuwien.ac.at*. Retrieved 2020-02-13.
14. Balyo, Tomáš; Sinz, Carsten (2018), "Parallel Satisfiability", *Handbook of Parallel Constraint Reasoning*, Springer International Publishing, pp. 3–29, doi:10.1007/978-3-319-63516-3_1 (https://doi.org/10.1007%2F978-3-319-63516-3_1), ISBN 978-3-319-63515-6
15. Biere, Armin. "Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010" (https://baldur.iti.kit.edu/sat-race-2010/descriptions/solver_1+2+3+6.pdf) (PDF). *SAT-RACE 2010*.
16. "ppfolio solver" (http://www.cril.univ-artois.fr/~roussel/ppfolio/). *www.cril.univ-artois.fr*. Retrieved 2019-12-29.
17. "SAT 2011 Competition: 32 cores track: ranking of solvers" (http://www.cril.univ-artois.fr/SAT11/results/ranking.php?idev=58). *www.cril.univ-artois.fr*. Retrieved 2020-02-13.
18. Balyo, Tomáš; Sanders, Peter; Sinz, Carsten (2015), "HordeSat: A Massively Parallel Portfolio SAT Solver", *Theory and Applications of Satisfiability Testing -- SAT 2015*, Lecture Notes in Computer Science, vol. 9340, Springer International Publishing, pp. 156–172, arXiv:1505.03340 (https://arxiv.org/abs/1505.03340), doi:10.1007/978-3-319-24318-4_12 (https://doi.org/10.1007%2F978-3-319-24318-4_12), ISBN 978-3-319-24317-7, S2CID 11507540 (https://api.semanticscholar.org/CorpusID:11507540)
19. "SAT Competition 2018" (http://sat2018.forsyte.tuwien.ac.at/). *sat2018.forsyte.tuwien.ac.at*. Retrieved 2020-02-13.
20. Heule, Marijn J. H.; Kullmann, Oliver; Wieringa, Siert; Biere, Armin (2012), "Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads", *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, vol. 7261, Springer Berlin Heidelberg, pp. 50–65, doi:10.1007/978-3-642-34188-5_8 (https://doi.org/10.1007%2F978-3-642-34188-5_8), ISBN 978-3-642-34187-8
21. Heule, Marijn J. H.; van Maaren, Hans (2009). "Look-Ahead Based SAT Solvers" (https://www.cs.utexas.edu/~marijn/publications/p01c05_lah.pdf) (PDF). *Handbook of Satisfiability*. IOS Press. pp. 155–184. ISBN 978-1-58603-929-5.
22. Heule, Marijn J. H.; Kullmann, Oliver; Marek, Victor W. (2016), "Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer", *Theory and Applications of Satisfiability Testing – SAT 2016*, Lecture Notes in Computer Science, vol. 9710, Springer International Publishing, pp. 228–245, arXiv:1605.00723 (https://arxiv.org/abs/1605.00723), doi:10.1007/978-3-319-40970-2_15 (https://doi.org/10.1007%2F978-3-319-40970-2_15), ISBN 978-3-319-40969-6, S2CID 7912943 (https://api.semanticscholar.org/CorpusID:7912943)
23. Ahmed, Tanbir (2010). "Two new van der Waerden numbers w(2;3,17) and w(2;3,18)". *Integers*. **10** (4): 369–377. doi:10.1515/integ.2010.032 (https://doi.org/10.1515%2Finteg.2010.032). MR 2684128 (https://mathscinet.ams.org/mathscinet-getitem?mr=2684128). S2CID 124272560 (https://api.semanticscholar.org/CorpusID:124272560).

24. Roli, Andrea (2002), "Criticality and Parallelism in Structured SAT Instances", *Principles and Practice of Constraint Programming - CP 2002*, Lecture Notes in Computer Science, vol. 2470, Springer Berlin Heidelberg, pp. 714–719, doi:10.1007/3-540-46135-3_51 (https://doi.org/10.1007%2F3-540-46135-3_51), ISBN 978-3-540-44120-5
25. Arbelaez, Alejandro; Hamadi, Youssef (2011), "Improving Parallel Local Search for SAT", *Learning and Intelligent Optimization*, Lecture Notes in Computer Science, vol. 6683, Springer Berlin Heidelberg, pp. 46–60, doi:10.1007/978-3-642-25566-3_4 (https://doi.org/10.1007%2F978-3-642-25566-3_4), ISBN 978-3-642-25565-6, S2CID 14735849 (https://api.semanticscholar.org/CorpusID:14735849)
26. Schöning, Uwe (Oct 1999). "A probabilistic algorithm for k-SAT and constraint satisfaction problems" (http://homepages.cwi.nl/~rdewolf/schoning99.pdf) (PDF). *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. pp. 410–414. doi:10.1109/SFFCS.1999.814612 (https://doi.org/10.1109%2FSFFCS.1999.814612). ISBN 0-7695-0409-4. S2CID 123177576 (https://api.semanticscholar.org/CorpusID:123177576).
27. "An improved exponential-time algorithm for k-SAT" (http://dl.acm.org/citation.cfm?id=1066101), Paturi, Pudlak, Saks, Zani
28. "Faster k-SAT algorithms using biased-PPSZ" (http://dl.acm.org/citation.cfm?id=3316359), Hansen, Kaplan, Zamir, Zwick
29. Kouril, Michal; Paul, Jerome L. (2008). "The van der Waerden Number $W(2,6)$ Is 1132" (https://projecteuclid.org/journals/experimental-mathematics/volume-17/issue-1/The-van-der-Waerden-Number-W26-Is-1132/em/1227031896.full). *Experimental Mathematics*. **17** (1): 53–61. doi:10.1080/10586458.2008.10129025 (https://doi.org/10.1080%2F10586458.2008.10129025). ISSN 1058-6458 (https://search.worldcat.org/issn/1058-6458). S2CID 1696473 (https://api.semanticscholar.org/CorpusID:1696473).
30. Kouril, Michal (2012). "Computing the van der Waerden number W(3,4)=293". *Integers*. **12**: A46. MR 3083419 (https://mathscinet.ams.org/mathscinet-getitem?mr=3083419).
31. Heule, Marijn J. H.; Kullmann, Oliver; Marek, Victor W. (2016), "Solving and Verifying the Boolean Pythagorean Triples Problem via Cube-and-Conquer", *Theory and Applications of Satisfiability Testing – SAT 2016*, Lecture Notes in Computer Science, vol. 9710, pp. 228–245, arXiv:1605.00723 (https://arxiv.org/abs/1605.00723), doi:10.1007/978-3-319-40970-2_15 (https://doi.org/10.1007%2F978-3-319-40970-2_15), ISBN 978-3-319-40969-6, S2CID 7912943 (https://api.semanticscholar.org/CorpusID:7912943)
32. Lamb, Evelyn (2016-06-01). "Two-hundred-terabyte maths proof is largest ever" (https://doi.org/10.1038%2Fnature.2016.19990). *Nature*. **534** (7605): 17–18. Bibcode:2016Natur.534...17L (https://ui.adsabs.harvard.edu/abs/2016Natur.534...17L). doi:10.1038/nature.2016.19990 (https://doi.org/10.1038%2Fnature.2016.19990). ISSN 1476-4687 (https://search.worldcat.org/issn/1476-4687). PMID 27251254 (https://pubmed.ncbi.nlm.nih.gov/27251254). S2CID 5528978 (https://api.semanticscholar.org/CorpusID:5528978).
33. "Schur Number Five" (https://www.cs.utexas.edu/~marijn/Schur/). *www.cs.utexas.edu*. Retrieved 2023-10-26.
34. Clarke, Edmund; Biere, Armin; Raimi, Richard; Zhu, Yunshan (2001-07-01). "Bounded Model Checking Using Satisfiability Solving" (https://doi.org/10.1023/A:1011276507260). *Formal Methods in System Design*. **19** (1): 7–34. doi:10.1023/A:1011276507260 (https://doi.org/10.1023%2FA%3A1011276507260). ISSN 1572-8102 (https://search.worldcat.org/issn/1572-8102). S2CID 2484208 (https://api.semanticscholar.org/CorpusID:2484208).
35. Biere, Armin; Cimatti, Alessandro; Clarke, Edmund M.; Strichman, Ofer; Zhu, Yunshan (2003). "Bounded Model Checking" (https://www.cs.cmu.edu/~emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model%20Checking.pdf) (PDF). *Advances in Computers*. **58** (2003): 117–148. doi:10.1016/S0065-2458(03)58003-2 (https://doi.org/10.1016%2FS0065-2458%2803%2958003-2). ISBN 9780120121588 – via Academic Press.

36. De Moura, Leonardo; Bjørner, Nikolaj (2011-09-01). "Satisfiability modulo theories: introduction and applications" (https://doi.org/10.1145/1995376.1995394). *Communications of the ACM*. **54** (9): 69–77. doi:10.1145/1995376.1995394 (https://doi.org/10.1145%2F1995376.1995394). ISSN 0001-0782 (https://search.worldcat.org/issn/0001-0782). S2CID 11621980 (https://api.semanticscholar.org/CorpusID:11621980).
37. Coelho, José; Vanhoucke, Mario (2011-08-16). "Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers" (https://www.sciencedirect.com/science/article/pii/S037722171100230X). *European Journal of Operational Research*. **213** (1): 73–82. doi:10.1016/j.ejor.2011.03.019 (https://doi.org/10.1016%2Fj.ejor.2011.03.019). ISSN 0377-2217 (https://search.worldcat.org/issn/0377-2217).
38. Peters, Dominik (2021). "Proportionality and Strategyproofness in Multiwinner Elections". arXiv:2104.08594 (https://arxiv.org/abs/2104.08594) [cs.GT (https://arxiv.org/archive/cs.GT)].
39. Brandl, Florian; Brandt, Felix; Peters, Dominik; Stricker, Christian (2021-07-18). "Distribution Rules Under Dichotomous Preferences: Two Out of Three Ain't Bad" (https://doi.org/10.1145/3465456.3467653). *Proceedings of the 22nd ACM Conference on Economics and Computation*. EC '21. New York, NY, USA: Association for Computing Machinery. pp. 158–179. doi:10.1145/3465456.3467653 (https://doi.org/10.1145%2F3465456.3467653). ISBN 978-1-4503-8554-1. S2CID 232109303 (https://api.semanticscholar.org/CorpusID:232109303).

External links

- Overview (http://www.satcompetition.org) of Sat competitions since 2002