CNFgen: A Generator of Crafted Benchmarks

Massimo Lauria[1(✉)], Jan Elffers[2], Jakob Nordström[2], and Marc Vinyals[2]

[1] Università degli studi di Roma "La Sapienza", Rome, Italy
massimo.lauria@uniroma1.it
[2] KTH Royal Institute of Technology, 100 44 Stockholm, Sweden

Abstract. We present `CNFgen`, a generator of combinatorial benchmarks in DIMACS and OPB format. The proof complexity literature is a rich source not only of hard instances but also of instances that are theoretically easy but "extremal" in different ways, and therefore of potential interest in the context of SAT solving. Since most of these formulas appear not to be very well known in the SAT community, however, we propose `CNFgen` as a resource to make them readily available for solver development and evaluation. Many formulas studied in proof complexity are based on graphs, and `CNFgen` is also able to generate, parse and do basic manipulation of such objects. Furthermore, it includes a library `cnfformula` giving access to the functionality of `CNFgen` to Python programs.

1 Introduction

The Boolean satisfiability problem (SAT) is a foundational problem in computational complexity theory. It was the first problem proven NP-complete [21], and is widely believed to be completely infeasible to solve in the worst case—indeed, a popular starting point for many other impossibility results in computational complexity theory is the *Exponential Time Hypothesis (ETH)* [33] postulating that there are no subexponential-time algorithms for SAT.

From an applied perspective SAT looks very different, however. In the last 15–20 years there has been a dramatic increase in the performance of satisfiability algorithms, or *SAT solvers*, and so-called *conflict-driven clause learning (CDCL) solvers* [5,37,41] are now routinely used to solve real-world instances with hundreds of thousands or even millions of variables.

Surprisingly, although the performance of current state-of-the-art SAT solvers is very impressive indeed, our understanding of *why* they work so well (at least most of the time) leaves much to be desired. Essentially the only known rigorous method for analysing SAT solvers is to use tools from *proof complexity* [22] to study the potential and limitations of the methods of reasoning they use.

The basic CDCL algorithm searches for *resolution proofs* [12]. Some solvers such as *PolyBoRi* [14,15] use algebraic *Gröbner basis computations*, but it seems hard to make them competitive with resolution-based solvers. A compromise is to have *Gaussian elimination* inside a resolution-based solver as in [30,48].

Webpage: https://massimolauria.github.io/cnfgen/.

S. Gaspers and T. Walsh (Eds.): SAT 2017, LNCS 10491, pp. 464–473, 2017.
DOI: 10.1007/978-3-319-66263-3_30

The power of these algebraic methods is captured by the *polynomial calculus (PC)* proof system [1,20]. There are also *pseudo-Boolean solvers* such as [18,24,35,47] exploring the geometric proof system *cutting planes (CP)* [23], although again it seems like a tough challenge to make these solvers as efficient as CDCL. We refer to the survey [42] and references therein for a more detailed discussion about the connections between proof complexity and SAT solving.

It seems fair to say that research in proof complexity into the proof systems mentioned above has not yielded too much by way of interesting insights for applied SAT solving so far. This is natural, since this research is driven mainly by theoretical concerns in computational complexity theory. However, what this body of work has produced is a wide selection of combinatorial formulas with interesting properties, and these we believe could be fruitfully mined for insights by SAT practitioners. As the SAT community starts to focus not only on producing blisteringly fast SAT solvers, but also on understanding better why these SAT solvers work the way they do, we expect that a study of combinatorial benchmarks could be particularly useful.

This immediately raises a question, however: Why do we need more crafted SAT problems? Is there really a need for more combinatorial benchmarks on top of what is already available in the standard SAT competition benchmarks?

We believe the answer is an emphatic "yes." In fact, it is our feeling that the SAT community has made quite limited use of crafted benchmarks so far. Most of these benchmarks are known to be dead hard for the resolution proof system, and will hence quickly grow out of reach of any CDCL solver (except if these solvers have dedicated preprocessing techniques to deal with such formulas, such as cardinality detection or Gaussian reasoning, but even then further minor tweaks to the benchmarks can easily make them infeasible).

This does not seem to be very informative—these benchmarks are hard simply because the method of reasoning employed by CDCL solvers cannot solve them efficiently in principle. A more interesting question is how well SAT solvers perform *when there are short proofs to be found*, and the solvers therefore have the potential to run fast. Studying solvers performance on such benchmarks can shed light on the quality of proof search, and indicate potential for improvement.

As a case in point, for the first time (to the best of our knowledge) many of the crafted benchmarks used in the *SAT Competition 2016* [4] (and generated by `CNFgen`) had the property that they possess extremely short resolution proofs and that SAT solvers can even be guided to find these proofs by, e.g., simply following a good fixed variable decision order. Yet the competition results showed that many of these benchmarks were beyond reach of even the best solvers.

It would seem that such formulas that are easy in theory for resolution but hard in practice for CDCL would merit further study if we want to understand what makes CDCL solvers fast and how they can be improved further, and `CNFgen` is a convenient tool for providing such formulas. An obvious downside is that such benchmarks can appear to be somewhat artificial in that one would not really run into them while solving applied problems. We readily concede this point. However, these formulas have the very attractive property that they

can be scaled freely to yield instances of different sizes—as opposed to applied benchmarks, that typically exist for a fixed size—and running the solvers on instances from the same family while varying the instance size makes it possible to tease out the true asymptotic behaviour.

By judiciously choosing formulas with different theoretical properties one can "stress-test" CDCL solvers on memory management (using formulas with size-space trade-off properties), restart policy (for formulas that are hard for strict subsystems of resolution), decision heuristic (for formulas that are easy with a good fixed variable order), et cetera, as done, e.g., in [26,34].

Furthermore, even theoretically hard crafted benchmarks can yield interesting insights in that they can be used to compare SAT solvers based on different methods of reasoning, for instance by benchmarking CDCL against algebraic solvers on formulas that are hard for resolution but easy for algebraic methods of reasoning, or against pseudo-Boolean solvers on formulas easy for cutting planes. `CNFgen` has been heavily used in work on analysing pseudo-Boolean solvers [25,52], which has so far generated quite intriguing and counter-intuitive results. (In particular, state-of-the-art pseudo-Boolean solvers sometimes struggle hopelessly with instances that are dead *easy* for the cutting planes method which they use to search for proofs, as also confirmed by benchmarks submitted to the *Pseudo-Boolean Competition 2016* [43].)

The `CNFgen` tool generates all of the CNF formulas discussed above in the standard DIMACS and OPB formats, thus making these benchmarks accessible to the applied SAT community. The included Python library allows formulas construction and manipulation, useful when encoding problems in SAT.

In Sect. 2 we present a small selection of the benchmarks in `CNFgen` and in Sect. 3 we illustrate some of its features. Concluding remarks are in Sect. 4.

2 Some Formula Families in `CNFgen`

A formula generator is a Python function that outputs a CNF, given parameters. A CNF is represented in our `cnfformula` library as a sequence of constrains (e.g., clauses, linear constraints, ...) defined over a set of named variables. `CNFgen` command line tool is essentially a wrapper around the available generators and the others CNF manipulation and SAT solving utilities in `cnfformula`.

Let us now describe briefly some examples of formulas available in `CNFgen`. Due to space constraints we are very far from giving a full list, and since new features are continuously being added such a list would soon be incomplete anyway. Typing `cnfgen --help` shows the full list of available formulas. The command `cnfgen <name> <params>` generates a formula from the family `<name>`, where the descriptions of the parameters needed is shown by `cnfgen <name> --help`.

Pigeonhole principle formulas (`php`) claim that m pigeons can be placed in n separate holes, where the variable $x_{i,j}$ encodes that pigeon i flies to hole j and the indices range over all $i \in [m]$ and $j \in [n]$ below. *Pigeon clauses* $\bigvee_{j=1}^{n} x_{i,j}$ enforce that every pigeon goes to a hole, and *hole clauses* $\overline{x}_{i,j} \vee \overline{x}_{i',j}$ for $i < i'$

forbid collisions. One can optionally include *functionality clauses* $\overline{x}_{i,j} \vee \overline{x}_{i,j'}$ for $j < j'$ and/or *onto clauses* $\bigvee_{i=1}^{m} x_{i,j}$ specifying that the mapping is one-to-one and onto, respectively. PHP formulas are unsatisfiable if and only if $m > n$ and if so require exponentially long proofs for all variants in resolution [29,44]. Functional onto-PHP formulas are easy for polynomial calculus (PC) but the other versions are hard (at least for a linear number of pigeons $m = O(n)$) [40]. All versions are easy for cutting planes (CP).

Tseitin formulas (`tseitin`) encode linear equation systems over GF(2) generated from connected graphs $G = (V, E)$ with *charge function* $\chi : V \to \{0, 1\}$. Edges $e \in E$ are identified with variables x_e, and for every vertex $v \in V$ we have the equation $\sum_{e \ni v} x_e \equiv \chi(v) \pmod 2$ encoded in CNF, yielding an unsatisfiable formula if and only if $\sum_{v \in V} \chi(v) \not\equiv 0 \pmod 2$. When G has bounded degree and is well-connected, the formula is hard for resolution [50] and for PC over fields of characteristic distinct from 2 [16], but is obviously easy if one can do Gaussian elimination (as in PC over GF(2)). Such Tseitin formulas are also believed to be hard for CP, but this is a major open problem in proof complexity. For long, narrow grid graphs, Tseitin formulas exhibit strong time-space trade-offs for resolution and PC [6,7].

Ordering principle formulas (`op`) assert that there is a partial ordering $\preceq$ of the finite set $\{e_1, \ldots, e_n\}$ so that no element is minimal, where variables $x_{i,j}$, $i \neq j \in [n]$, encode $e_i \preceq e_j$. Clauses $\overline{x}_{i,j} \vee \overline{x}_{j,i}$ and $\overline{x}_{i,j} \vee \overline{x}_{j,k} \vee x_{i,k}$ for distinct $i, j, k \in [n]$ enforce asymmetry and transitivity, and the non-minimality claim is encoded as clauses $\bigvee_{i \in [n] \setminus \{j\}} x_{i,j}$ for every $j \in [n]$. The **total ordering principle** also includes clauses $x_{i,j} \vee x_{j,i}$ specifying that the order is total. The **graph ordering principle** (`gop`) is a "sparse version" where the non-minimality of e_j must be witnessed by a neighbour e_i in a given graph (which for the standard version is the complete graph). For well-connected graphs these formulas are hard for DPLL but easy for resolution [13,49]. If the well-connected graphs are sparse, so that all initial clauses have bounded size, the formulas have the interesting property that any resolution or PC proof must still contain clauses/polynomials of large size/degree [13,27].

Random k-CNF formulas (`randkcnf`) with m clauses over n variables are generated by randomly picking m out of the $2^k\binom{n}{k}$ possible k-literal clauses without replacement. These formulas are unsatisfiable with high probability for $m = \Delta_k \cdot n$ with Δ_k a large enough constant depending on k, where $\Delta_2 = 1$ (provably) and $\Delta_3 \approx 4.26$ (empirically). Random k-CNFs for $k \geq 3$ are hard for resolution and PC [3,19] and most likely also for CP, although this is again a longstanding open problem.

Pebbling formulas (`peb`) are defined in terms of directed acyclic graphs (DAGs) $G = (V, E)$, with vertices $v \in V$ identified with variables x_v, and contain clauses saying that (a) source vertices s are true (a unit clause x_s) and

(b) truth propagates through the DAG (clauses $\bigvee_{i=1}^{\ell} \overline{x}_{u_i} \vee x_v$ for each non-source v with predecessors $u_1, \ldots, u_\ell$) but (c) sinks z are false (a unit clause $\overline{x}_z$). Pebbling formulas are trivially refuted by unit propagation, but combined with transformations as described in Sect. 3 they have been used to prove time-space trade-offs for resolution, PC, and CP [7–9,32] and have also been investigated from an empirical point of view in [34].

Stone formulas (`stone`) are similar to pebbling formulas, but here each vertex of the DAG contains a stone, where (a) stones on sources are red and (b) a non-source with all predecessors red also has a red stone, but (c) sinks have blue stones. This unsatisfiable formula has been used to separate general resolution from so-called *regular* resolution [2] and has also been investigated when comparing the power of resolution and CDCL without restarts [17].

k-clique formulas (`kclique`) declare that a given graph $G = (V, E)$ has a k-clique. Variables $x_{i,v}$, $i \in [k]$, $v \in V$, constrained by $\sum_{v \in V} x_{i,v} = 1$ identify k vertices, and clauses $\overline{x}_{i,u} \vee \overline{x}_{j,v}$ for every non-edge $\{u, v\} \notin E$ and $i \neq j \in [k]$ enforce that these vertices form a clique. For k constant it seems plausible that their proof length should scale roughly like $|V|^k$ in the worst case but this remains wide open even for resolution and only partial results are known [10,11].

Subset cardinality formulas (`subsetcard`). For a 0/1 $n \times n$ matrix $A = (a_{i,j})$, identify positions where $a_{i,j} = 1$ with variables $x_{i,j}$. Letting $R_i = \{j \mid a_{i,j} = 1\}$ and $C_j = \{i \mid a_{i,j} = 1\}$ record the positions of 1s/variables in row i and column j, the formula encodes the cardinality constraints $\sum_{j \in R_i} x_{i,j} \geq |R_i|/2$ and $\sum_{i \in C_j} x_{i,j} \leq |C_i|/2$ for all $i, j \in [n]$. In the case when all rows and columns have $2k$ variables, except for one row and column that have $2k + 1$ variables, the formula is unsatisfiable but is hard for resolution and polynomial calculus if the positions of the variables are "scattered enough" (such as when M is the bipartite adjacency matrix of an expander graph) [39,51]. Cutting planes, however, can just add up all constraints to derive a contradiction immediately.

Even colouring formulas (`ec`) are defined on connected graphs $G = (V, E)$ with all vertices having bounded, even degree. Edges $e \in E$ correspond to variables x_e, and for all vertices $v \in V$ constraints $\sum_{e \ni v} x_e = \deg(v)/2$ assert that there is a 0/1-colouring such that each vertex has an equal number of incident 0- and 1-edges. The formula is satisfiable if and only if the total number of edges is even. For suitably chosen graphs these formulas are empirically hard for CDCL [36], but we do not know of any formal resolution lower bounds. Despite being easy for CP, they still seem hard for pseudo-Boolean solvers.

3 Further Tools for CNF Generation and Manipulation

Formula transformations. A common trick to obtain hard proof complexity benchmarks is to take a CNF formula and replace each variable x by a Boolean

function $g(x_1, \ldots, x_\ell)$ of arity ℓ over new variables. As an example, XOR substitution $y \leftarrow y_1 \oplus y_2$, $z \leftarrow z_1 \oplus z_2$ applied to the clause $y \vee \overline{z}$ yields

$$(y_1 \vee y_2 \vee z_1 \vee \overline{z}_2) \wedge (y_1 \vee y_2 \vee \overline{z}_1 \vee z_2) \wedge \ (\overline{y}_1 \vee \overline{y}_2 \vee z_1 \vee \overline{z}_2) \wedge (\overline{y}_1 \vee \overline{y}_2 \vee \overline{z}_1 \vee z_2) \ .$$

Note that such transformations can dramatically increase formula size, and so they work best when the size of the initial clauses and the arity ℓ is small. Similar substitutions, and also other transformations such as *lifting*, *shuffling*, and *variable compression* from [45], can be applied either in `CNFgen` during formula generation (using command line options `-T`), or alternatively to a DIMACS file using the included `cnftransform` program. Multiple occurrences of `-T <params>` results in a chain of transformations as in, e.g., this 2-xorified pebbling formula over the pyramid graph of height 10, with random shuffling.

```
$ cnfgen peb --pyramid 10  -T xor 2 -T shuffle
```

Formulas based on graphs. Many formulas in `CNFgen` are generated from graphs, which can be either read from a file or produced internally by the tool. In the next example we build a Tseitin formula over the graph in the file `G.gml` and then a graph ordering principle on a random 3-regular graph with 10 vertices.

```
$ cnfgen tseitin -i G.gml --charge randomodd | minisat
UNSATISFIABLE
$ cnfgen gop --gnd 10 3 | minisat
UNSATISFIABLE
```

The `CNFgen` command line provides some basic graph constructions and also accepts graphs in different formats such as, e.g., Dot [46], DIMACS [38], and GML [31]. Inside Python there is more flexibility since any NetworkX [28] graph object can be used, as sketched in the next example.

```
from cnfformula import GraphColoringFormula
G= ...   # build the graph
GraphColoringFormula(G,4).dimacs() # Is G is 4-colourable?
```

As already discussed in Sect. 2, the hardness of many formulas generated from graphs are governed by (different but related notions of) *graph expansion*. Going into details is beyond the scope of this paper, but in many cases a randomly sampled regular graph of bounded vertex degree almost surely has the expansion required to yield hard instances.

OPB output format. `CNFgen` supports the OPB format used by pseudo-Boolean solvers, which use techniques based on cutting planes. `CNFgen` can produce formulas that are easy for cutting planes but seem quite hard for pseudo-Boolean solvers (e.g., subset cardinality formulas, even colouring formulas, some kinds of k-colouring instances).

4 Concluding Remarks

We propose `CNFgen` as a convenient tool for generating crafted benchmarks in DIMACS or OPB. `CNFgen` makes available a rich selection of formulas appearing in the proof complexity literature, and new formulas can easily be added by using the `cnfformula` library. It is our hope that this tool can serve as something of a one-stop shop for, e.g., SAT practitioners wanting to benchmark their solvers on tricky combinatorial formulas, competition organizers looking for crafted instances, proof complexity researchers wanting to test theoretical predictions against actual experimental results, and mathematicians performing theoretical research by reducing to SAT.

Acknowledgments. The first author performed most of this work while at KTH Royal Institute of Technology. The authors were funded by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement no. 279611 as well as by Swedish Research Council grant 621-2012-5645. The first author was also supported by the European Research Council under the European Union's Horizon 2020 Research and Innovation Programme / ERC grant agreement no. 648276 AUTAR.

References

1. Alekhnovich, M., Ben-Sasson, E., Alexander, A., Razborov, A.A., Wigderson, A.: Space complexity in propositional calculus. SIAM J. Comput. **31**(4), 1184–1211 (2002). Preliminary version in STOC '00
2. Alekhnovich, M., Johannsen, J., Pitassi, T., Urquhart, A.: An exponential separation between regular and general resolution. Theor. Comput. **3**(5), 81–102 (2007). Preliminary version in STOC '02
3. Alekhnovich, M., Alexander, A. Razborov, A.A.: Lower bounds for polynomial calculus: Non-binomial case. In: Proceedings of the Steklov Institute of Mathematics, 242, 18–35 (2003). http://people.cs.uchicago.edu/~razborov/files/misha.pdf. Preliminary version in FOCS '01
4. Balyo, T., Marijn, J., Heule, H., Järvisalo, M.: Proceedings of SAT competition 2016: Solver and benchmark descriptions. Technical report B-2016-1, University of Helsinki (2016). http://hdl.handle.net/10138/164630
5. Roberto, J., Bayardo, Jr., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997), pp. 203–208 (1997)
6. Beame, P., Beck, C., Impagliazzo, R.: Time-space tradeoffs in resolution: Superpolynomial lower bounds for superlinear space. In: Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC 2012), pp. 213–232 (2012)
7. Beck, C., Nordström, J., Tang, B.: Some trade-off results for polynomial calculus. In: Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC 2013), pp. 813–822 (2013)
8. Ben-Sasson, E., Nordström, J.: Short proofs may be spacious: An optimal separation of space and length in resolution. In: Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2008), pp. 709–718 (2008)

9. Ben-Sasson, E., Nordström, J.: Understanding space in proof complexity: Separations and trade-offs via substitutions. In: Proceedings of the 2nd Symposium on Innovations in Computer Science (ICS 2011), pp. 401–416 (2011)
10. Beyersdorff, O., Galesi, N., Lauria, M.: Parameterized complexity of DPLL search procedures. ACM Trans. Comput. Logic **14**(3), 20:1–20:21 (2013). Preliminary version in SAT '11
11. Beyersdorff, O., Galesi, N., Lauria, M., Razborov, A.A.: Parameterized bounded-depth Frege is not optimal. ACM Trans. Comput. Theor. **4**, 7:1–7:16 (2012). Preliminary version in ICALP '11
12. Blake, A.: Canonical expressions in boolean algebra. PhD thesis, University of Chicago (1937)
13. Bonet, M.L., Galesi, N.: Optimality of size-width tradeoffs for resolution. Comput. Complex. **10**(4), 261–276 (2001). Preliminary version in FOCS '99
14. Brickenstein, M., Dreyer, A.: PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials. J. Symb. Comput. **44**(9), 1326–1345 (2009)
15. Brickenstein, M., Dreyer, A., Greuel, G.-M., Wedler, M., Wienand, O.: New developments in the theory of Gröbner bases and applications to formal verification. J. Pure Appl. Algebr. **213**(8), 1612–1635 (2009)
16. Buss, S.R., Grigoriev, D., Impagliazzo, R., Pitassi, T.: Linear gaps between degrees for the polynomial calculus modulo distinct primes. J. Comput. Syst. Sci. **62**(2), 267–289 (2001). Preliminary version in CCC '99
17. Buss, S.R., Kołodziejczyk, L.: Small stone in pool. Logic. Method. Comput. Sci. **10**, 10:16–16:22 (2014)
18. Chai, D., Kuehlmann, A.: A fast pseudo-Boolean constraint solver. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **24**(3), 305–317 (2005). Preliminary version in DAC '03
19. Chvátal, V., Szemerédi, E.: Many hard examples for resolution. J. ACM **35**(4), 759–768 (1988)
20. Clegg, M., Edmonds, J., Impagliazzo, R.: Using the Groebner basis algorithm to find proofs of unsatisfiability. In: Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC 1996), pp. 174–183 (1996)
21. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151–158 (1971)
22. Cook, S.A., Reckhow, R.: The relative efficiency of propositional proof systems. J. Symbol. Logic **44**(1), 36–50 (1979)
23. Cook, W.: Collette Rene Coullard, and György Turán. On the complexity of cutting-plane proofs. Discr. Appl. Math. **18**(1), 25–38 (1987)
24. Dixon, H.E., Ginsberg, M.L., Hofer, D.K., Luks, E.M., Parkes, A.J.: Generalizing Boolean satisfiability III: Implementation. J. Artif. Intell. Res. **23**, 441–531 (2005)
25. Elffers, J., Giráldez-Crú, J., Nordström, J., Vinyals, M.: Using combinatorial benchmarks to probe the reasoning power of pseudo-Boolean solvers (2017, Submitted)
26. Elffers, J., Nordström, J., Simon, L., Sakallah, K.A.: Seeking practical CDCL insights from theoretical SAT benchmarks. In: Presentation at the Pragmatics of SAT 2016 workshop (2016). http://www.csc.kth.se/~jakobn/research/TalkPoS16.pdf
27. Galesi, N., Lauria, M.: Optimality of size-degree trade-offs for polynomial calculus. ACM Trans. Comput. Logic **12**, 4:1–4:22 (2010)

28. Hagberg, A.A., Schult, D., Swart, P.S.: Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008), Pasadena, CA USA, pp. 11–15 (2008)
29. Haken, A.: The intractability of resolution. Theor. Comput. Sci. **39**(2–3), 297–308 (1985)
30. Heule, M., van Maaren, H.: Aligning CNF- and equivalence-reasoning. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 145–156. Springer, Heidelberg (2005). doi:10.1007/11527695_12
31. Himsolt, M.: GML: A portable graph file format. Technical report, Universität of Passau (1996)
32. Huynh, T., Nordström, J.: On the virtue of succinct proofs: Amplifying communication complexity hardness to time-space trade-offs in proof complexity (Extended abstract). In: Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC 2012), pp. 233–248 (2012)
33. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001). Preliminary version in CCC '99
34. Järvisalo, M., Matsliah, A., Nordström, J., Živný, S.: Relating proof complexity measures and practical hardness of SAT. In: Milano, M. (ed.) CP 2012. LNCS, pp. 316–331. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33558-7_25
35. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. J. Satisf. Boolean Model. Comput. **7**, 59–64 (2010)
36. Markström, K.: Locality and hard SAT-instances. J. Satisf. Boolean Model. Comput. **2**(1–4), 221–227 (2006)
37. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999). Preliminary version in ICCAD '96
38. Massey, B.: DIMACS graph format (2001). http://prolland.free.fr/works/research/dsat/dimacs.html. Accessed 11 Feb 2016
39. Mikša, M., Nordström, J.: Long proofs of (seemingly) simple formulas. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 121–137. Springer, Cham (2014). doi:10.1007/978-3-319-09284-3_10
40. Mikša, M., Nordström, J.: A generalized method for proving polynomial calculus degree lower bounds. In: Proceedings of the 30th Annual Computational Complexity Conference (CCC 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 33, pp. 467–487 (2015)
41. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Chaff, M.S.: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535 (2001)
42. Nordström, J.: On the interplay between proof complexity and SAT solving. ACM SIGLOG News **2**(3), 19–44 (2015)
43. Pseudo-Boolean competition (2016). http://www.cril.univ-artois.fr/PB16/
44. Razborov, A.A.: Resolution lower bounds for perfect matching principles. J. Comput. Syst. Sci. **69**(1), 3–27 (2004). Preliminary version in CCC '02
45. Razborov, A.A.: A new kind of tradeoffs in propositional proof complexity. J. ACM **63**, 16:1–16:14 (2016)
46. AT and T Research: Dot Language. http://www.graphviz.org/content/dot-language. Accessed 11 Feb 2016
47. Sheini, H.M., Sakallah, K.A.: Pueblo: a hybrid pseudo-Boolean SAT solver. J. Satisf. Boolean Model. Comput. **2**(1–4), 165–189 (2006). Preliminary version in DATE '05

48. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_24
49. Stålmarck, G.: Short resolution proofs for a sequence of tricky formulas. Acta Inform. **33**(3), 277–280 (1996)
50. Urquhart, A.: Hard examples for resolution. J. ACM **34**(1), 209–219 (1987)
51. Gelder, A., Spence, I.: Zero-one designs produce small hard SAT instances. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 388–397. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_37
52. Vinyals, M., Elffers, J., Giráldez-Crú, J., Gocht, S., Nordström, J.: In between resolution and cutting planes: A study of proof systems for pseudo-Boolean SAT solving (2017, Submitted)