A generator of hard 2QBF formulas and ASP programs

Giovanni Amendola[1], **Francesco Ricca**[1], **Miroslaw Truszczynski**[2]
[1]Department of Mathematics and Computer Science, University of Calabria, Italy
{amendola,ricca}@mat.unical.it
[2]Department of Computer Science, University of Kentucky, USA
mirek@cs.uky.edu

Abstract

The availability of generators of random instances of boolean formulas has had a major impact on solver technology for KR formalisms such as SAT, QBF and ASP. Recently, we proposed new models of random QSAT formulas in non-clausal form, as well as the first model of random disjunctive logic programs. Our models support generating instances of substantial hardness. Here, we present a tool that generates formulas/programs from the new models in a variety of output formats including (Q)DIMACS, QCIR, and ASPCore 2.0.

Introduction

The ability to generate large numbers of formulas of a desired hardness is important for many KRR formalisms, and in particular it received much attention in SAT and QBF communities (Gent and Walsh 1999). Indeed, inherently hard instances are essential for designing and testing new search techniques (Achlioptas 2009), and are used in solver competitions (Järvisalo et al. 2012a; Narizzano, Pulina, and Tacchella 2006). Large collections of *easy* instances support the so-called *fuzz* testing and can help reveal issues in solver implementation, as well as defects in solver design (Brummayer, Lonsing, and Biere 2010). Random formulas have been used successfully to assess CDCL solvers (Silva, Lynce, and Malik 2009) on resource management and efficacy of heuristics (Elffers et al. 2016; Järvisalo et al. 2012b). The ability to generate large numbers of easy and hard logic programs is equally important to the field of Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011) for developing and testing solvers (Alviano et al. 2015; Calimeri et al. 2016; Gebser et al. 2007; Gebser, Maratea, and Ricca 2017b).

The tools to generate formulas and programs for solver development and testing often resort to random generation (Creignou, Egly, and Seidl 2012; Lauria et al. 2017). Accordingly, class of formulas (hereafter, referred to as models) of random formulas and programs have received substantial attention from the artificial intelligence community in the last twenty years (Achlioptas 2009; Mitchell, Selman, and Levesque 1992; Selman, Mitchell, and Levesque 1996). In this paper we present an instance generation tool that generates instances from the models of random formulas, QBFs and disjunctive logic programs recently proposed by Amendola, Ricca, and Truszczynski (2017; 2018). The tool can also generate CNF formulas from the well-known fixed-length model (Mitchell, Selman, and Levesque 1992), and QBFs from the Chen-Interian model (Chen and Interian 2005). The tool supports standard output formats used in SAT, QBF and ASP competitions: (Q)DIMACS (Järvisalo et al. 2012a; Narizzano, Pulina, and Tacchella 2006), QCIR (Jordan, Klieber, and Seidl 2016), and ASPCore 2.0 (Calimeri et al. 2016).

Random CNF formulas, QBFs, and programs

In this section, we recall the *fixed-length clause model* for random CNF formulas and the *Chen-Interian model* for random QBF formulas. We then describe the models of QBFs and programs proposed by Amendola et al. (2017; 2018).

By $C(k,n,m)$ we denote the set of all k-CNF formulas consisting of m clauses over (some fixed) set of n propositional variables. Similarly, $D(k,n,m)$ stands for the set of all k-DNF formulas of m *products* (conjunctions of non-complementary literals) over an n-element set of atoms.

The fixed-length clause model. The model is given by the set $C(k,n,m)$ of CNF formulas, with all formulas assumed equally likely.

The Chen-Interian model. The model generates QBFs of the form $\forall X \exists Y F$, where sets X and Y are disjoint, $|X| = A$, $|Y| = E$, and F is a CNF formula with m clauses, each containing a literals with variables in X and e literals with variables in Y. We write $Q(a,e;A,E;m)$ for the set of all such QBFs. The Chen-Interian model generates QBFs from $Q(a,e;A,E;m)$, with all formulas equally likely.

The controlled model. Also this model generates QBFs of the form $\forall X \exists Y F$, where sets X and Y are disjoint, $|X| = A$, $|Y| = E$, and F is a CNF formula. However, now F has $2A$ clauses, one for each literal with a variable in X, with all remaining literals in the clause with variables in Y. We write $Q^{ctd}(k,A,E)$ for the the set of such QBFs. The *controlled* model generates QBFs from $Q^{ctd}(k,A,E)$, with all formulas equally likely. Clearly, $Q^{ctd}(k,A,E) \subseteq Q(1,k-1;A,E;2A)$. Thus, the controlled model is related to the Chen-Interian model. The main difference is that the clauses, while random with respect to existential variables are not random with

respect to universal variables.

The multi-component model. This construct can be applied to any model considered above. Let $\mathscr{F}$ be a class of propositional formulas. By t-$\mathscr{F}$ we denote the class of all disjunctions of t formulas from $\mathscr{F}$. Similarly, if $\mathscr{Q}$ is a class of QBFs of the form $\forall X \exists Y F$, where $F \in \mathscr{F}$, we write t-$\mathscr{Q}$ for the class of all QBFs of the form $\forall X \exists Y F$, where $F \in t$-$\mathscr{F}$. We refer to models t-$\mathscr{F}$ and t-$\mathscr{Q}$ as *multi-component*. We assume that formulas (QBFs, respectively) in the model are equally likely.

Models of random logic programs. Every QBF $\exists X \forall Y F$, where F is a DNF formula can be translated by the translation of Eiter and Gottlob (Eiter and Gottlob 1995) to a disjunctive logic program so that true QBFs are mapped to consistent programs. The Chen-Interian and the controlled models described above have their dual counterparts. Thus, each gives rise to the corresponding model of disjunctive logic programs. We denote them $D_{dlp}(e,a;E,A;m)$ and $D^{ctd}_{dlp}(E,A,k)$, respectively. The Eiter-Gottlob translation has a simple extension to QBFs obtained from the multi-component models. Thus, multi-component Chen-Interian and controlled models imply multi-component models of programs denoted by t-$D_{dlp}(e,a;E,A;m)$ and t-$D^{ctd}_{dlp}(E,A,k)$, respectively.

Phase Transition and Hardness. The fixed-length clause model exhibits a well-known *easy-hard-easy* pattern (Mitchell, Selman, and Levesque 1992) (sometimes more accurately called the "easy-hard-less hard" pattern (Coarfa et al. 2000)). It is aligned with the phase transition area, where the satisfiability of formulas switches from SAT to UNSAT. Experimental results show that formulas prior to the phase transition are "easy" to solve. At the phase transition, they are "hard" to solve. Finally, past the phase transition they are again "easy" to solve (some prefer to say "less hard" as they are typically harder than those prior to the phase transition). All models discussed above have similar satisfiability properties, showing a phase transition region and an easy-hard-easy pattern (Chen and Interian 2005; Amendola et al. 2017; 2018).

Implementation

We developed our tool in Java. Its modular architecture supports a variety of generation methods and output formats employed in SAT, QBF and ASP competitions: (Q)DIMACS (Järvisalo et al. 2012a; Narizzano, Pulina, and Tacchella 2006), QCIR (Jordan, Klieber, and Seidl 2016), and ASPCore 2.0 (Calimeri et al. 2016).

All formats ensure that clauses do not contain complementary literals, conjunctions of clauses/rules contain no repetition of the same clause/rule, and formulas/programs generated according to the multi-component model contain no repetitions of the same component. Formulas appearing in our multi-component models are non-clausal (when $t > 1$), whereas the (Q)DIMACS format requires the formulas to be in CNF. We implemented Tseitin's translation of non-clausal theories to CNF (Tseitin 1983). It is worth noting that the translation for multi-component logic programs is particularly simple (Amendola et al. 2018). The tool is implemented to be used as Posix command line command, but could be integrated as a library of a more general purpose random generation environment.[1]

```
$ java -jar RandomGenerator.jar -h

SYNOPSIS: MainGenerator [-option]
-generator=[BasicGenerator,CIGenerator,SATGenerator,
           ControlledCIGenerator] Select generator type
-out=[PrintProgram,PrintQBF,PrintQCIR,MultiOutput,
      PrintSAT] select output format
-o=<filename> Specify filename, mandatory with
              MultiOutputGenerator, default STDOUT
-formats=<OutputFormat1, ..., OutputFormatn>
 Specify a comma separated list of output formats for
 MultiOutput, e.g., PrintProgram,PrintQBF
-E=<n> Number of existential variables, default 1
-A=<n> Number of universal variables, default 1
-c=<n> Number of clauses/rules,
       ignored by ControlledCIGenerator, default 1
-k=<n> Clause/rule size only for BasicGenerator, default 1
-e=<n> Number of existentials in each clause/rule
       only for CI, default 1
-a=<n> Number of universals in each clause/rule
       only for CI, default 1
-w=<n> Number of components, default 1
```

Figure 1: Command line and help message of the generator.

Usage

We now describe how to invoke the tool from the command line, provide guidelines for generating hard formulas, and report on the tool's known applications.

Invocation. Figure 1 shows the result of running the tool with option *-h*. It prints an explanation of the available options. In particular, the generation model can be selected by specifying a value for *-generator* option: *BasicGenerator* corresponds to QBFs in the fixed-length clause model, *CIGenerator* corresponds to the Chen-Interian model, *ControlledCIGenerator* corresponds to the controlled model, and *SATGenerator* corresponds to SAT formulas in the fixed-length clause model.

Model generation can be paired with different output formats specified by option *-out*. The five main options are: *PrintProgram, PrintQBF, PrintQCIR, PrintSAT*, and *MultiOutput*. The first four options output logic programs in ASPCore 2.0, QBFs in QDIMACS, QBFs in QCIR, and CNF formulas in DIMACS formats, respectively. Setting *-out=MultiOutput* has the tool print the same instance in several output formats that can be specified with option *-format*. This is useful when one plans to run several solvers on the same sets of instances, eliminating the need for format converters. Once a formula is generated, it is stored in several files having as name the string provided as parameter of the *-o* option. Parameters such as the numbers of variables, clauses/rules, components and others can specified by using (some of) the options shown in Figure 1.

[1]The generator is available at: `https://www.mat.unical.it/ricca/RandomLogicProgramGenerator`.

```
$ java -jar RandomGenerator.jar -generator=CIGenerator
       -out=PrintQBF -o=10-CI-2-3-20-40 -w=10 -a=2 -e=3
       -A=20 -E=40 -c=80

$ java -jar RandomGenerator.jar
       -generator=ControlledCIGenerator
       -out=MultiOutputGenerator
       -formats=PrintProgram,PrintQBF,PrintQCIR
       -o=4-Qctd-4-20-10 -w=4 -a=1 -e=3 -A=20 -E=10
```

Figure 2: Two examples of invocations of the *RandomGenerator* tool from the command line.

Two examples are provided in Figure 2. The first command generates one instance of the class 10-$Q(2,3;20,40;80)$ and prints it in QDIMACS format. The instances of this class have 10 components (`-w=10`) that are Chen-Interian model formulas. Each formula has 80 clauses (`-c=80`), each clause has two universal (`-a=2`) and three existential (`-e=3`) variables. They are selected from a set of 20 universal (`-A=20`) and 40 existential variables (`-E=40`), respectively. The result is stored in a single file called `10-CI-2-3-20-40.dimacs`. The second command generates one instance in the class 4-$Q^{ctd}(4,20,10)$, and prints it in three formats: ASPCore 2.0, QDIMACS and QCIR. The instances of this class have four components (`-w=4`). They are formulas from the controlled model with clauses having one universal (`-a=1`) and three existential (`-e=3`) variables. They are selected from a set of 20 universals (`-A=20`) and 10 existential variables (`-E=10`), respectively. The result is stored in three files `4-Qctd-4-20-10` with extensions `.asp, .dimacs, .qcir`.

Generating formulas. The generator can be used for assessing solver performance or testing an implementation, and these activities usually have different requirements for instance properties. In presence of so many generation parameters, it is not obvious how to choose the right settings for the purpose. Here we provide simple guidelines for identifying the settings for generating formulas of the desired hardness. The key underlying property is that all our models show some form of the easy-hard-easy pattern that can be exploited to find "areas" of formulas of varying difficulty.

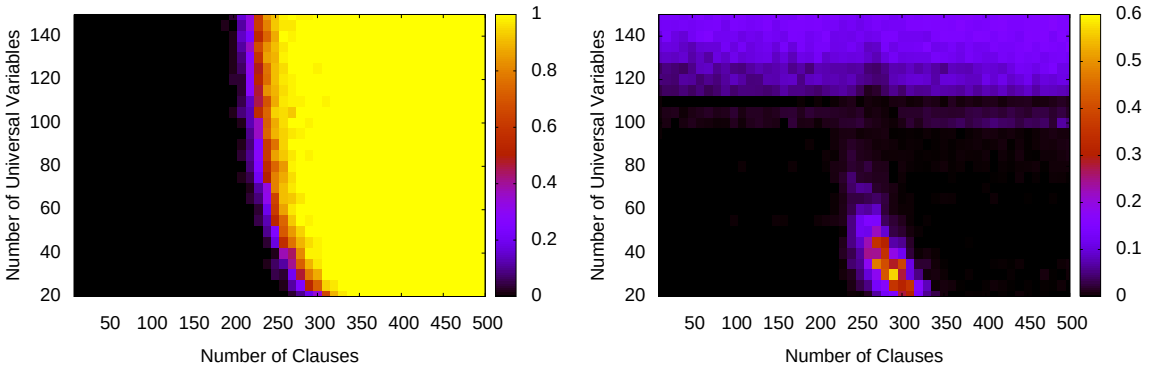
For the Chen-Interian model, one strategy is to fix a and e (to define the structure of a clause). Next for each pair of values of $A = |X|$ and $E = |Y|$, one runs the tool with different numbers m of clauses/rules. The formulas (programs) generated in this way show the phase transition and the corresponding easy-hard-easy pattern. Running a solver on those formulas allows one to observe these properties and select the value of m that yields formulas/programs of the desired difficulty. Figure 3 shows the results of this study for the Chen-Interian model instances with 128 samples for each size and $a = 1$, $e = 3$ and, for (a-a'), $E = 50$ and $t = 1$, for (b-b') $E = 60$ and $t = 1$, and for (c-c'), $E = 60$ and $t = 2$. In each case, we let A vary from 20 to 140, and for each value of A we ranged m from 50 to 500. The results show the phase transition (plotting the number of SAT instances for each sample, i.e., the frequency of SAT) and its correlations with the easy-hard-easy pattern (plotting average execution time per sample). Comparing Figure 3 (a') with Figure 3 (b') we note that hardness (expectedly) increases with E; whereas comparing Figure 3 (b-b') with Figure 3 (c-c') we observe that the phase transition shifts left and hardness *significantly* grows with the number of components, as explained by Amendola et al. (2017; 2018). Thus, increasing t one can get *super-hard* instances with fewer variables w.r.t. standard models. These properties are solver-independent (Amendola et al. 2017; 2018), which was demonstrated by experiments with several well-known SAT solvers (Audemard, Lagniez, and Simon 2013; Biere 2014; Dequen and Dubois 2006), QBF solvers (Heule et al. 2015; Pigorsch and Scholl 2010; Janota et al. 2016), and ASP solvers (Alviano et al. 2015; Gebser et al. 2007). Moreover, when $t \geq 2$ multi-component models generate instances better solved by solvers for real-world instances (Amendola et al. 2017; 2018), which is a desirable property for testing/assessing CDCL-based solvers (Ansótegui, Bonet, and Levy 2009; Giráldez-Cru and Levy 2016).

Results in Figure 3 are a starting point for fine-tuning the parameters to obtain the desired hardness/satisfiability property. We developed similar strategies for the Controlled model and obtained analogous systematic pictures. These results are available with the distribution of our tool.

Known uses. Instances produced by the generator tool have been recently used in system competitions for QBF (Pulina 2016; Pulina and Seidl 2017) and Answer Set Programming (ASP) (Gebser, Maratea, and Ricca 2017a), yielding some of the hardest instances in these events.

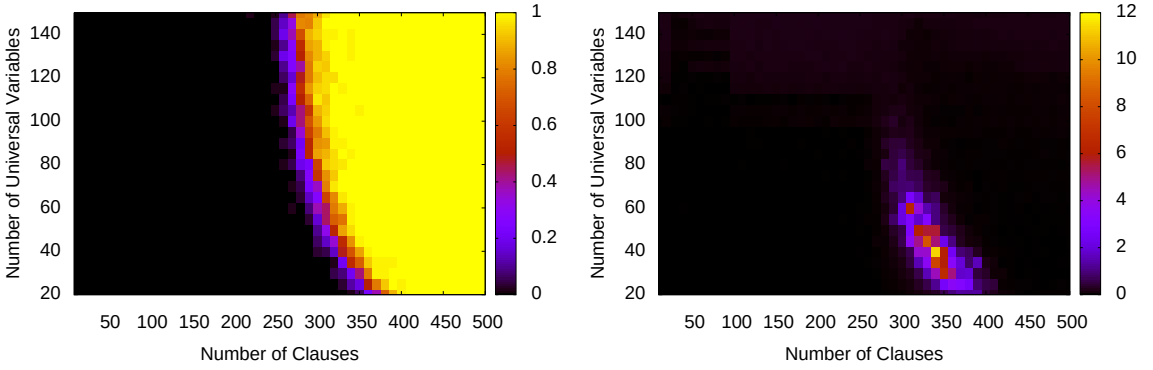
Three sets of instances were submitted to QBF Evaluations, see (Pulina and Seidl 2017). The instance set "Model instances" used in 2016 comprised 60 formulas from the t-component Chen-Interian model with 40 universal and 60 existential variables, and with t ranging from two to six, sampled in the hard region (the number of clauses varying from 200 to 432). None of these instances was classified as easy (solved by all 21 solvers), about 28% was classified as *hard* (no solver could solve any of them in 600s), about 7% *medium-hard* (only one solver could solve any of them), and about 65% *medium* (otherwise). In 2017 we submitted two sets of instances of the multi-component controlled model, labeled "Selection hard" and "wgrowing" in QBF Evalaution. The first set contained 10 instances with 48 existential variables, up to 146 universal variables, and up to 9 components. Half of these instances were tagged *hard* and the other half *medium*. The second set contained 33 instances from the hard region sampled from a set of 256 instances generated uniformly with 48 existential variables and 128 universal variables by varying the number of components from 3 to 9. In this set 73% were tagged *hard* and 27% were tagged *medium*. Interestingly, our instances helped identify a bug in one of the participating solvers, showing the efficacy of our model also in correctness testing.

We submitted a set of 128 instances of the multi-component controlled model, labeled "Random Disjunctive Programs" to the 2017 ASP competition. These instances were generatd for 80 universal variables, 24 existential variables, and 9 components. The organizers classified 121 instances as *hard* (solved by only one reference solver), 2 instances as *very hard* (solved by no reference solver) and the



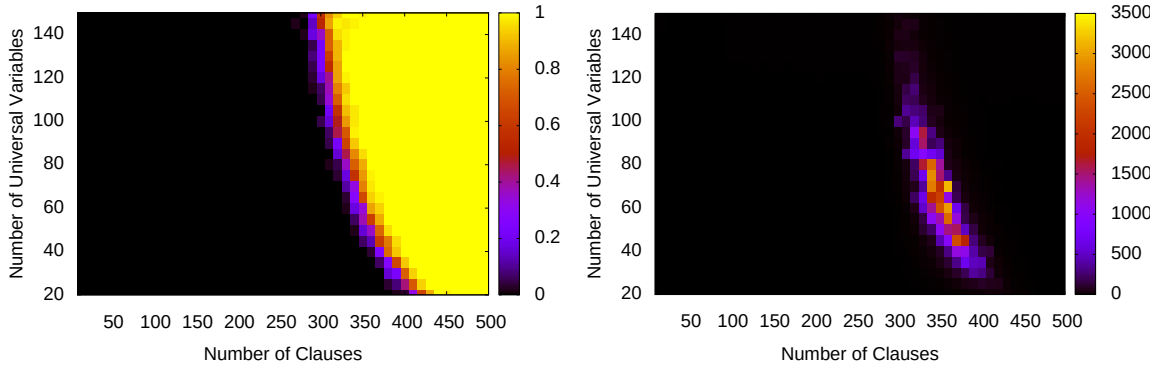
(a) Frequency of SAT for 1-$Q(1,3,A,50,m)$

(a') Execution Time (s) for 1-$Q(1,3,A,50,m)$



(b) Frequency of SAT for 1-$Q(1,3,A,60,m)$

(b') Execution Time (s) for 1-$Q(1,3,A,60,m)$



(c) Frequency of SAT for 2-$Q(1,3,A,60,m)$

(c') Execution Time (s) for 2-$Q(1,3,A,60,m)$

Figure 3: Phase transition and hardness in (multicomponent) Chen-Interian formulas.

remaining 5 as *easy* (solved by all solvers). For the competition, the organizers selected 20 of these instances. No solver could solve all these instances. Importantly, they were the smallest in size but among the hardest to solve in the entire event in the search category (Gebser, Maratea, and Ricca 2017a). We could submit even harder instances (simply increasing variables or components), but providing a too-hard instance set would not be beneficial for comparing solvers.

Conclusions

We described a generator of fixed-length clause, Chen-Interian, controlled and multi-component models for random disjunctive logic programs and QSAT formulas. The tool supports a variety of standard output formats such as (Q)DIMACS, QCIR, and ASPCore 2.0, and provides practical means for generating new benchmarks to evaluate robustness and performance of SAT, QBF and ASP solvers. The tool has already been used in several competitions that showed its ability to generate formulas that can challenge even the best solvers, especially when a combination of the controlled and multi-component model is employed.

Acknowledgments

The third author was partially supported by the NSF grant IIS-1707371; the first two were partially supported by MISE under project S^{2}BDW n. F/050389/01-02-03/X32.

References

Achlioptas, D. 2009. Random satisfiability. In *Handbook of Satisfiability*. 245–270.

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015. Advances in WASP. In *LPNMR 2015*, 40–54.

Amendola, G.; Ricca, F.; and Truszczynski, M. 2017. Generating hard random boolean formulas and disjunctive logic programs. In *Proceedings of IJCAI 2017*.

Amendola, G.; Ricca, F.; and Truszczynski, M. 2018. New models for generating hard random boolean formulas and disjunctive logic programs. *CoRR* abs/1802.03828v1.

Ansótegui, C.; Bonet, M. L.; and Levy, J. 2009. Towards industrial-like random SAT instances. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 387–392.

Audemard, G.; Lagniez, J.; and Simon, L. 2013. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *SAT 2013*, 309–317.

Biere, A. 2014. Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver lingeling. In *Proc. of POS-14.*, volume 27 of *EPiC Series*, 88.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM* 54(12):92–103.

Brummayer, R.; Lonsing, F.; and Biere, A. 2010. Automated testing and debugging of SAT and QBF solvers. In *SAT 2010*, 44–57.

Calimeri, F.; Gebser, M.; Maratea, M.; and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artif. Intell.* 231:151–181.

Chen, H., and Interian, Y. 2005. A model for generating random quantified boolean formulas. In Kaelbling, L. P., and Saffiotti, A., eds., *IJCAI 2005*, 66–71. Professional Book Center.

Coarfa, C.; Demopoulos, D. D.; Aguirre, A. S. M.; Subramanian, D.; and Vardi, M. Y. 2000. Random 3-sat: The plot thickens. In *CP 2000*, 143–159.

Creignou, N.; Egly, U.; and Seidl, M. 2012. A framework for the specification of random SAT and QSAT formulas. In *TAP 2012*, 163–168.

Dequen, G., and Dubois, O. 2006. An efficient approach to solving random *k*-satproblems. *J. Autom. Reasoning* 37(4):261–276.

Eiter, T., and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.* 15(3-4):289–323.

Elffers, J.; Nordström, J.; Simon, L.; and Sakallah, K. 2016. Seeking practical cdcl insights from theoretical sat benchmarks. In *Presentation at the Pragmatics of SAT 2016 workshop*.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. *clasp* : A conflict-driven answer set solver. In *LPNMR 2007*, 260–265.

Gebser, M.; Maratea, M.; and Ricca, F. 2017a. The design of the seventh answer set programming competition. In *LPNMR*, volume 10377 of *Lecture Notes in Computer Science*, 3–9. Springer.

Gebser, M.; Maratea, M.; and Ricca, F. 2017b. The sixth answer set programming competition. *J. Artif. Intell. Res.* 60:41–95.

Gent, I. P., and Walsh, T. 1999. Beyond NP: the QSAT phase transition. In *AAAI 1999*, 648–653.

Giráldez-Cru, J., and Levy, J. 2016. Generating SAT instances with community structure. *Artif. Intell.* 238:119–134.

Heule, M.; Järvisalo, M.; Lonsing, F.; Seidl, M.; and Biere, A. 2015. Clause elimination for SAT and QSAT. *J. Artif. Intell. Res. (JAIR)* 53:127–168.

Janota, M.; Klieber, W.; Marques-Silva, J.; and Clarke, E. M. 2016. Solving QBF with counterexample guided refinement. *Artif. Intell.* 234:1–25.

Järvisalo, M.; Berre, D. L.; Roussel, O.; and Simon, L. 2012a. The international SAT solver competitions. *AI Magazine* 33(1).

Järvisalo, M.; Matsliah, A.; Nordström, J.; and Zivny, S. 2012b. Relating proof complexity measures and practical hardness of SAT. In *CP*, volume 7514 of *Lecture Notes in Computer Science*, 316–331. Springer.

Jordan, C.; Klieber, W.; and Seidl, M. 2016. Non-cnf qbf solving with QCIR. In *Proceedings AAAI-16 Workshop on Beyond NP*, to appear.

Lauria, M.; Elffers, J.; Nordström, J.; and Vinyals, M. 2017. Cnfgen: A generator of crafted benchmarks. In *SAT*, volume 10491 of *Lecture Notes in Computer Science*, 464–473. Springer.

Mitchell, D. G.; Selman, B.; and Levesque, H. J. 1992. Hard and easy distributions of SAT problems. In *AAAI 1992*, 459–465.

Narizzano, M.; Pulina, L.; and Tacchella, A. 2006. Report of the third QBF solvers evaluation. *JSAT* 2(1-4):145–164.

Pigorsch, F., and Scholl, C. 2010. An aig-based qbf-solver using SAT for preprocessing. In *DAC 2010*, 170–175. ACM.

Pulina, L., and Seidl, M. 2017. Qbf eval 2017 `http://www.qbflib.org/qbfeval17.php`.

Pulina, L. 2016. The ninth QBF solvers evaluation - preliminary report. In *QBF@SAT*, volume 1719 of *CEUR Workshop Proceedings*, 1–13. CEUR-WS.org.

Selman, B.; Mitchell, D. G.; and Levesque, H. J. 1996. Generating hard satisfiability problems. *Artif. Intell.* 81(1-2):17–29.

Silva, J. P. M.; Lynce, I.; and Malik, S. 2009. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 131–153.

Tseitin, G. 1983. On the complexity of derivation in propositional calculus. In Siekmann, J. H., and Wrightson, G., eds., *Automation of Reasoning*, Symbolic Computation. Springer Berlin Heidelberg. 466–483.