Artificial Intelligence 243 (2017) 1–25

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

The Configurable SAT Solver Challenge (CSSC) ☆

Frank Hutter [a,*], Marius Lindauer [a], Adrian Balint [b], Sam Bayless [c], Holger Hoos [c], Kevin Leyton-Brown [c]

[a] University of Freiburg, Germany
[b] University of Ulm, Germany
[c] University of British Columbia, Vancouver, Canada

ARTICLE INFO

Article history:
Received 8 May 2015
Received in revised form 16 July 2016
Accepted 28 September 2016
Available online 20 October 2016

Keywords:
Propositional satisfiability
Algorithm configuration
Empirical evaluation
Competition

ABSTRACT

It is well known that different solution strategies work well for different types of instances of hard combinatorial problems. As a consequence, most solvers for the propositional satisfiability problem (SAT) expose parameters that allow them to be customized to a particular family of instances. In the international SAT competition series, these parameters are ignored: solvers are run using a single default parameter setting (supplied by the authors) for all benchmark instances in a given track. While this competition format rewards solvers with robust default settings, it does not reflect the situation faced by a practitioner who only cares about performance on one particular application and can invest some time into tuning solver parameters for this application. The new Configurable SAT Solver Competition (CSSC) compares solvers in this latter setting, scoring each solver by the performance it achieved after a fully automated configuration step. This article describes the CSSC in more detail, and reports the results obtained in its two instantiations so far, CSSC 2013 and 2014.

1. Introduction

The propositional satisfiability problem (SAT) is one of the most prominent problems in AI. It is relevant both for theory (having been the first problem proven to be NP-hard [27]) and for practice (having important applications in many fields, such as hardware and software verification [19,72,26], test-case generation [79,24], AI planning [53,54], scheduling [28], and graph coloring [83]). The SAT community has a long history of regularly assessing the state of the art via competitions [50]. The first SAT competition dates back to the year 2002 [76], and the event has been growing over time: in 2014, it had a record participation of 58 solvers by 79 authors in 11 tracks [13].

In practical applications of SAT, solvers can typically be adjusted to perform well for the specific type of instances at hand, such as software verification instances generated by a particular static checker on a particular software system [3], or a particular family of bounded model checking instances [85]. To support this type of customization, most SAT solvers already expose a range of command line *parameters* whose settings substantially affect most parts of the solver. Solvers typically come with robust default parameter settings meant to provide good all-round performance, but it is widely known that

☆ This paper was submitted to the Competition Section of the journal.

* Corresponding author.
E-mail addresses: fh@cs.uni-freiburg.de (F. Hutter), lindauer@cs.uni-freiburg.de (M. Lindauer), adrian.balint@uni-ulm.de (A. Balint), sbayless@cs.ubc.ca (S. Bayless), hoos@cs.ubc.ca (H. Hoos), kevinlb@cs.ubc.ca (K. Leyton-Brown).

http://dx.doi.org/10.1016/j.artint.2016.09.006
0004-3702/

adjusting parameter settings to particular target instance classes can yield orders-of-magnitude speedups [42,55,81]. Current SAT competitions do not take this possibility of customizing solvers into account, and rather evaluate solver performance with default parameters.

Unlike the SAT competition, the *Configurable SAT Solver Challenge (CSSC)* evaluates SAT solver performance *after* application-specific customization, thereby taking into account the fact that effective algorithm configuration procedures can automatically customize solvers for a given distribution of benchmark instances. Specifically, for each type of instances T and each SAT solver S, an automated fixed-time offline configuration phase determines parameter settings of S optimized for high performance on T. Then, the performance of S on T is evaluated with these settings, and the solver with the best performance wins.

To avoid a potential misunderstanding, we note that for winning the competition, only solver performance after configuration counts, and that it does *not* matter how much performance was improved by configuration. As a consequence, in principle, even a parameterless solver could win the CSSC if it was very strong: it would not benefit from configuration, but if it nevertheless outperformed all solvers that were specially configured for the instance families in a given track, it would still win that track. (In practice, we have not observed this, since the improvements resulting from configuration tend to be large.)

The competition conceptually most closely related to the CSSC is the learning track of the international planning competition (IPC, see, e.g., the description by Fern et al. [31],[1] which also features an offline time-limited learning phase on training instances from a given planning domain and an online testing phase on a disjoint set of instances from the same domain. The main difference between this IPC learning track and the CSSC (other than their focus on different problems) is that in the IPC learning track every planner uses its own learning method, and the learning methods thus vary between entries. In contrast, in the CSSC, the corresponding customization process is part of the competition setup and uses the same algorithm configuration procedure for each submitted solver. Our approach to evaluating solver performance after configuration could of course be transferred to any other competition. (In fact, the 2014 IPC learning track for non-portfolio solvers was won by FastDownward-*SMAC* [75], a system that employs a similar combination of general algorithm configuration and a highly parameterized solver framework as we do in the CSSC.)

In the following, we first describe the criteria we used for the design of the CSSC (Section 2). Next, we provide some background on the automated algorithm configuration methods we used when running the competition (Section 3). Then, we discuss the two CSSCs we have held so far (in 2013 and 2014); we discuss each of these competitions in turn (Sections 4 and 5), including the specific benchmarks used, the participating solvers, and the results. We describe two main insights that we obtained from these results:

1. In many cases, automated algorithm configuration found parameter settings that performed much better than the solver defaults, in several cases yielding average speedups of several orders of magnitude.
2. Some solvers benefited more from automated configuration than others; as a result, the ranking of algorithms after configuration was often substantially different from the ranking based on the algorithm defaults (as, e.g., measured in the SAT competition).

Finally, we analyze various aspects of these results (Section 6) and discuss the implications we see for future algorithm development (Section 7).

2. Design criteria for the CSSC

We organized the CSSC 2013 and 2014 in coordination with the international SAT competition and presented them in the competition slots at the 2013 and 2014 SAT conferences (as well as in the 2014 FLoC Olympic Games, in which all SAT-related competitions took part). We coordinated solver submission deadlines with the SAT competition to minimize overhead for participants, who could submit their solver to the SAT competition using default parameters and then open up their parameter spaces for the CSSC.

We designed the CSSC to remain close to the international SAT competition's established format; in particular, we used the same general categories: *industrial*, *crafted*, and *random*, and, in 2014 also *random satisfiable*. Furthermore, we used the same input and output formats, the SAT competition's mature code for verifying correctness of solver outputs (only for checking models of satisfiable instances; we did not have a certified UNSAT track), and the same scoring function (number of instances solved, breaking ties by average runtime).

The main way our setup differed from that of the SAT competition was that we used a relatively small budget of five minutes per solver run. We based this choice partly on the fact that many solvers have runtime distributions with rather long tails (or even heavy tails [35]), and that practitioners often use *many instances* and relatively *short runtimes* to benchmark solvers for a new application domain. There is also evidence that SAT competition results would remain quite similar if based on shorter runtimes, but not if based on fewer instances [44]. Therefore, in order to achieve more robust performance within a fixed computational budget, we chose to use many test instances (at least 250 for each benchmark)

[1] http://www.cs.colostate.edu/~ipc2014/.

but relatively low runtime cutoffs per solver run (five minutes). (We also note that a short time limit of five minutes has already been used in the agile track of the 2014 International Planning Competition.) Due to constraints imposed by our computational infrastructure, we used a memory limit of 3GB for each solver run.

To simulate the situation faced by practitioners with limited computational resources, we limited the computational budget for configuring a solver on a benchmark with a given configuration procedure to two days on 4 or 5 cores (in 2014 and 2013, respectively). Our results are therefore indicative of what could be obtained by performing configuration runs over the weekend on a modern desktop machine.

2.1. Controlled execution of solver runs

Since all configuration procedures ran in an entirely automated fashion, they had to be robust against any kind of solver failure (segmentation faults, unsupported combinations of parameters, wrong results, infinite loops, etc.). We handled all such conditions in a generic wrapper script that used Olivier Roussel's runsolver tool [73] to limit runtime and memory, and counted any errors or limit violations as timeouts at the maximum runtime of 300 seconds. We also kept track of the rich solver runtime data we gathered in our configuration runs and made it publicly available on the competition website.

2.2. Choice of configuration pipeline

To avoid bias arising from our choice of algorithm configuration method, we independently used all three state-of-the-art methods applicable for runtime optimization (*ParamILS* [47], *GGA* [1], and *SMAC* [46], as described in detail in Section 3). We evaluated the configurations resulting from all configuration runs on the entire training data set and selected the configuration with the best training performance. We then executed only this configuration on the test set to determine the performance of the configured solver. Except where specifically noted otherwise, all performance data we report in this article is for this optimized configuration on previously unseen test instances from the respective benchmark set.

2.3. Pre-submission bug fixing

As part of the submission package, we provided solver authors with our configuration pipeline, so that they could run it themselves to identify bugs in their solver before submission (e.g., problems due to the choice of non-default parameters). We also provided some trivial benchmark sets for this pre-submission preparation, which were not part of the competition.

We did not offer a bug fixing phase after solver submission, except that we ran a very simple configuration experiment (10 minutes on trivial instances) to verify that the setup of all participants was correct.

2.4. Choice of benchmarks

We chose the benchmark families for the CSSC to be relatively homogeneous in terms of the origin and/or construction process of instances in the same family. Typically, we selected benchmark families that are neither too easy (since speedups are less interesting for easy instances), nor too hard (so that solvers could solve a large fraction of instances within the available computational budgets). We aimed for benchmark sets of which at least 20–40% could be solved within the maximum runtime on a recent machine by the default configuration of a SAT solver that would perform reasonably well in the SAT competition. We also aimed for benchmark sets with a sufficient number of instances to safeguard against over-tuning; in practice, the smallest datasets we used had 500 instances: 250 for training and 250 for testing.

We did not disclose which benchmark sets we used until the competition results were announced. While we encouraged competition entrants to also contribute benchmarks, we made sure to not substantially favor any solver by using such contributed benchmarks.

3. Automated algorithm configuration procedures

The problem of finding performance-optimizing algorithm parameter settings arises for many computational problems. In recent years, the AI community has developed several dedicated systems for this general *algorithm configuration* problem [47,1,57,46].

We now describe this problem more formally. Let A be an algorithm having n parameters with domains $\Theta_1, \ldots, \Theta_n$. Parameters can be *real-valued* (with domains $[a, b]$, where $a, b \in \mathbb{R}$ and $a < b$), *integer-valued* (with domains $[i, j]$, where $i, j \in \mathbb{Z}$ and $i < j$), or *categorical* (with finite unordered domains, such as {red, blue, green}). Parameters can also be *conditional* on an instantiation of other (so-called *parent*) parameters; as an example, consider the parameters of a heuristic mechanism h, which are completely ignored unless h is chosen to be used by means of another, categorical parameter. Finally, some combinations of parameter instantiations can be labeled as *forbidden*.

Algorithm A's *configuration space* $\boldsymbol{\Theta}$ then consists of all possible combinations of parameter values: $\boldsymbol{\Theta} = \Theta_1 \times \cdots \times \Theta_n$. We refer to elements $\boldsymbol{\theta} = \langle \theta_1, \ldots, \theta_n \rangle$ of this configuration space as *parameter configurations*, or simply *configurations*. Given a benchmark set Π and a performance metric $m(\boldsymbol{\theta}, \pi)$ capturing the performance of configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ on problem instance

$\pi \in \Pi$, the algorithm configuration problem then aims to find a configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ that minimizes m over Π, *i.e.*, that minimizes[2]

$$f(\boldsymbol{\theta}) = \frac{1}{|\Pi|} \cdot \sum_{\pi \in \Pi} m(\boldsymbol{\theta}, \pi).$$

In the CSSC, the specific metric m we optimized was *penalized average runtime (PAR-10)*, which counts runs that exceed a maximal cutoff time κ_{max} without solving the given instance as $10 \cdot \kappa_{max}$. We terminated individual solver runs as unsuccessful after $\kappa_{max} = 300$ seconds.

We refer to an instance of the algorithm configuration problem as a *configuration scenario* and to a method for solving the algorithm configuration problem as a *configuration procedure* (or a *configurator*), in order to avoid confusion with the solver to be optimized (which we refer to as the *target algorithm*) and the problem instances the solver is being optimized for.

Algorithm configuration has been demonstrated to be very effective for optimizing various SAT solvers in the literature. For example, Hutter et al. [42] configured the algorithm Spear [5] on formal verification instances, achieving a 500-fold speedup on software verification instances generated with the static checker Calysto [3] and a 4.5-fold speedup on IBM bounded model checking instances by Zarpas [85]. Algorithm configuration has also enabled the development of general frameworks for stochastic local search SAT solvers that can be automatically instantiated to yield state-of-the-art performance on new types of instances; examples for such frameworks are SATenstein [55] and Captain Jack [81].

While all of these applications used the local-search based algorithm configuration method *ParamILS* [47], in the CSSC we wanted to avoid bias that could arise from commitment to one particular algorithm configuration method and thus used all three existing general algorithm configuration methods for runtime optimization: *ParamILS*, *GGA* [1], and *SMAC* [46].[3] We refer the interested reader to Appendix B for details on each of these configurators. Here, we only mention some details that were important for the setup of the CSSC:

- *ParamILS* does not natively support parameters specified only as real- or integer-valued intervals, but requires all parameter values to be listed explicitly; for simplicity, we refer to the transformation used to satisfy this requirement as *discretization*. When multiple parameter spaces were available for a solver, we only ran *ParamILS* on the discretized version, whereas we ran *GGA* and *SMAC* on both the discretized and the non-discretized versions.
- *ParamILS* and *SMAC* have been shown to benefit substantially from multiple independent runs, since they are randomized algorithms. Given k cores, the usual approach is simply to execute k independent configurator runs and pick the configuration from the one with best performance on the training set. *GGA*, on the other hand, can use multiple cores on a single machine, and in fact requires these to run effectively. Therefore, given k available cores per configuration approach, we used k independent runs of each *ParamILS* and *SMAC*, and one run using all k cores for *GGA*.
- *GGA* could not handle the complex parameter conditionalities found in some solvers; for those solvers, we only ran *ParamILS* and *SMAC*.

4. The Configurable SAT Solver Challenge 2013

The first CSSC[4] was held in 2013. It featured three tracks mirroring those of the SAT competition: *Industrial SAT+UNSAT*, *crafted SAT+UNSAT*, and *Random SAT+UNSAT*. Table 1 lists the benchmark families we used in each of these tracks, all of which are described in detail in Appendix A. Within each track, we used the same number of test instances for each benchmark family, thereby weighting each equally in our analysis.

4.1. Participating solvers and their parameters

Table 2 summarizes the solvers that participated in the CSSC 2013, along with information on their configuration spaces. The eleven submitted solvers ranged from complete solvers based on conflict-directed clause learning (CDCL; [10]) to stochastic local search (SLS; [40]) solvers. The degree of parameterization varied substantially across these submitted solvers, from 2 to 241 parameters. We briefly discuss the main features of the solvers' parameter configuration spaces, ordering solvers by their number of parameters.

Gnovelty+GCa and *Gnovelty+GCwa* [29] are closely related SLS solvers. Both have two numerical parameters: the probability of selecting false clauses randomly and the probability of smoothing clause weights. The parameters were pre-discretized by the solver developer to 11 and 10 values, yielding 110 possible combinations.

[2] An alternative definition considers the optimization of expected performance across a *distribution* of instances rather than average performance across a *set* of instances [47]. What we consider here can be seen as a special case where the distribution is uniform over a given set of training instances. It is also possible to optimize performance metrics other than mean performance across instances, but mean performance is by far the most widely used option.

[3] We did not use the iterated racing method I/F-Race [57], since it does not effectively support runtime optimization and its authors thus discourage its use for this purpose (personal communication with Manuel López-Ibáñez and Thomas Stützle).

[4] http://www.cs.ubc.ca/labs/beta/Projects/CSSC2013/.

Table 1
Overview of benchmark sets used in the CSSC 2013 tracks *Industrial SAT+UNSAT*, *crafted SAT+UNSAT*, and *Random SAT+UNSAT* (from top to bottom); k and m stand for factors of one thousand and one million, respectively.

Benchmark	#Train	#Test	#Variables	#Clauses	Reference
SWV	302	302	68.9k ± 57.0k	182k ± 206k	[4]
IBM	383	302	96.4k ± 170k	413k ± 717k	[85]
Circuit Fuzz	299	302	5.53k ± 7.45k	18.8k ± 25.3k	[23]
BMC	807	302	446k ± 992k	1.09m ± 2.70m	[18]
GI	1032	351	11.2k ± 17.8k	2.98m ± 8.03m	[68,82]
LABS	350	351	75.9k ± 75.7k	154k ± 153k	[69]
K3	300	250	262 ± 43	1116 ± 182	[11]
unif-k5	300	250	50 ± 0	1056 ± 0	–
5sat500	250	250	500 ± 0	10000 ± 0	[81]

Table 2
Overview of solvers in the Configurable SAT Solver Challenge (CSSC) 2013 and their parameters of various types ('c' for categorical, 'i' for integer, 'r' for real-valued'); 'cond' identifies how many of these parameters are conditional. We also list the sizes of the configuration spaces provided by the solver developers (original, discretized, and the subset of the discrete parameters). Solvers are ordered by the total number of parameters they expose ($c + i + r$).

Solver	# Parameters				# Configurations			Reference
	c	i	r	Cond.	Original	Discretized	Disc. subset	
Gnovelty+GCa	2	0	0	0	110	–	–	[29]
Gnovelty+GCwa	2	0	0	0	110	–	–	[29]
Gnovelty+PCL	5	0	0	0	20 000	–	–	[29]
Simpsat	5	0	0	0	2 400	–	–	[36]
Sat4j	10	0	0	4	2×10^7	–	–	[14]
Solver43	12	0	0	0	5×10^6	–	–	[6]
Forl-nodrup	44	0	0	0	3×10^{18}	–	–	[78]
Clasp-2.1.3	42	34	7	60	∞	10^{45}	–	[33]
Riss3g	125	0	0	107	2×10^{53}	–	–	[63]
Riss3gExt	193	0	0	168	2×10^{82}	–	–	[63]
Lingeling	102	139	0	0	1×10^{974}	1×10^{136}	2×10^{39}	[16]

Gnovelty+PCL [29] is an SLS solver with five parameters: one binary parameter (determining whether the stagnation path is dynamic or static) and four numerical parameters: the length of the stagnation path, the size of the time window storing stagnation paths, the probability of smoothing stagnation weights, and the probability of smoothing clause weights. All numerical parameters were pre-discretized to ten values each by the solver developer, yielding 20 000 possible combinations.

Simpsat [36] is a CDCL solver based on *Cryptominisat* [77], which adds additional strategies for explicitly handling XOR constraints [37]. It has five numerical parameters that govern both these XOR constraint strategies and the frequency of random decisions. All parameters were pre-discretized by the solver developer, yielding 2 400 possible combinations.

Sat4j [14] is full-featured library of solvers for Boolean satisfiability and optimization problems. For the contest, it applied its default CDCL SAT solver with ten exposed parameters: four categorical parameters deciding between different restart strategies, phase selection strategies, simplifications, and cleaning; and six numerical parameters pre-discretized by its developer.

Solver43 [6] is a CDCL solver with 12 parameters: three categorical parameters concerning sorting heuristics used in bounded variable elimination, in definitions and in adding blocked clauses; and nine numerical parameters concerning various frequencies, factors, and limits. All parameters were pre-discretized by the solver developer.

Forl-nodrup [78] is a CDCL solver with 44 parameters. Most notably, these control variable selection, Boolean propagation, restarts, and learned clause removal. About a third of the parameters are numerical (particularly most of those concerning restarts and learned clause removal); all parameters were pre-discretized by the solver developer.

Clasp-2.1.3 [33] is a solver for the more general answer set programming (ASP) problem, but it can also solve SAT, MAXSAT and PB problems. As a SAT solver, *Clasp-2.1.3* is a CDCL solver with 83 parameters: 7 for pre-processing, 14 for the variable selection heuristic, 18 for the restart policy, 34 for the deletion policy, and 10 for a variety of other uses. The configuration space is highly conditional, with several top-level parameters enabling or disabling certain strategies. *Clasp-2.1.3* exposes both a mixed continuous/discrete parameter configuration space and a manually-discretized one.

Riss3g [63] is a CDCL solver with 125 parameters. These include 6 numerical parameters from MiniSAT [30], 10 numerical parameters from Glucose [2], 17 mostly numerical Riss3G parameters, and 92 parameters controlling preprocessing/inprocessing performed by the integrated Coprocessor [62]. The inprocessor parameters resemble those in *Lingeling* [16],

Table 3
Winners of the three tracks of CSSC 2013.

Rank	*Industrial SAT+UNSAT*	*Crafted SAT+UNSAT*	*Random SAT+UNSAT*
1st	*Lingeling*	*Clasp-3.0.4-p8*	*Clasp-3.0.4-p8*
2nd	*Riss3g*	*Forl-nodrup*	*Lingeling*
3rd	*Solver43*	*Lingeling*	*Riss3g*

emphasizing blocked clause elimination [51], bounded variable addition [65], and probing [61]. About 50 of the parameters are Boolean, and most others are numerical parameters pre-discretized by the solver developer. The parameter space is highly conditional, with inprocessor parameters dependent on a switch turning them on alongside various other dependencies. Indeed, there are only 18 unconditional parameters. Finally, there are also seven forbidden parameter combinations that ascertain various switches are turned on if inprocessing is used.

Riss3gExt [63] is an experimental extension of *Riss3g*. It exposes all of the parameters previously discussed for *Riss3g*, along with an additional 11 Riss3G parameters and 57 inprocessing parameters. Its developer implemented all of these extensions in one week and did not have time for extensive testing before the CSSC; therefore, he submitted *Riss3gExt* as closed source, making it ineligible for medals. We discuss the results of this closed-source solver separately, in Appendix C.

Lingeling [16] is a CDCL solver with 241 parameters (making it the solver with the largest configuration space in the CSSC 2013). 102 of these parameters are categorical, and the remaining 139 are integer-valued (76 of them with the trivial upper bound of max-integer, $2^{31} - 1$). *Lingeling* parameterizes many details of the solution process, including probing and look-ahead (about 25 mostly numerical parameters), blocked clause elimination and bounded variable elimination (about 20 mostly categorical parameters each), glue clauses (about 15 mostly numerical parameters), and a host of other mechanisms parameterized by about 5–10 parameters each. *Lingeling* exposes its full parameter space, a discretized version of all parameters, and a subspace of only the categorical parameters (102 of them).

4.2. Configuration pipeline

We executed this competition on the QDR partition of the Compute Canada Westgrid cluster Orcinus. Each node in this cluster was provisioned with 24 GB memory and two 6-core, 2.66 GHz Intel Xeon X5650 CPUs with 12 MB L2 cache each, and ran Red Hat Enterprise Linux Server 5.5 (kernel 2.6.18, glibc 2.5).

In this first edition of the CSSC, we were unfortunately unable to run *GGA*. This was because it requires multiple cores for effective runtime minimization, and the respective multiple-core jobs we submitted on the Orcinus cluster were stuck in the queue for months without getting started. (Single-core runs, on the other hand, were often scheduled within minutes.)

We thus limited ourselves to using *ParamILS* for the discretized parameter space of each of the 11 solvers and *SMAC* for each of the parameter spaces that solver authors submitted (as discussed above, 9 submissions with one parameter space, 1 submission with two, and 1 submission with three, i.e., 14 in total). For each of the nine benchmark families, this gave rise to 11 configuration scenarios for *ParamILS* and 14 for *SMAC*, for a total of 225 configuration scenarios. Since our budget for each configuration procedure was two CPU days on five cores (five independent runs of *ParamILS* and *SMAC*, respectively), the competition's configuration phase required a total of 2250 CPU days (just over 6 CPU years). Thanks to a special allocation on the Orcinus cluster, we were able to complete this phase within a week.

Following standard practice, we then evaluated the configurations resulting from all configuration runs on the entire training data set and selected the configuration with the best training performance. We then executed only this configuration on the test set to assess the performance of the configured solver. This evaluation phase required much less time than the configuration phase.

We note that all scripts we used for performing the configuration and analysis experiments were written in Ruby and are available for download on the competition website.

4.3. Results

For each the three tracks of CSSC 2013, we configured each of the eleven submitted solvers for each of the benchmark families within the track and aggregated results across the respective test instances. We show the winners in Table 3 and discuss the results for each track in the following sections. Additional details, tables, and figures are provided in an accompanying technical report [43].

We remind the reader that the CSSC score only depends on how well the configured solver did and *not* on the difference between default and configured performance. We nevertheless still cover default performance prominently in the following results, in order to emphasize the impact configuration had and the difference between the CSSC and standard solver competitions (e.g., the SAT competition).

4.3.1. Results of the Industrial SAT+UNSAT track

Our *Industrial SAT+UNSAT* track consisted of the four industrial benchmarks detailed in Appendix A.1: *Bounded Model Checking 2008 (BMC)* [15], *Circuit Fuzz* [23], *Hardware Verification (IBM)* [85], and *SWV* [4].

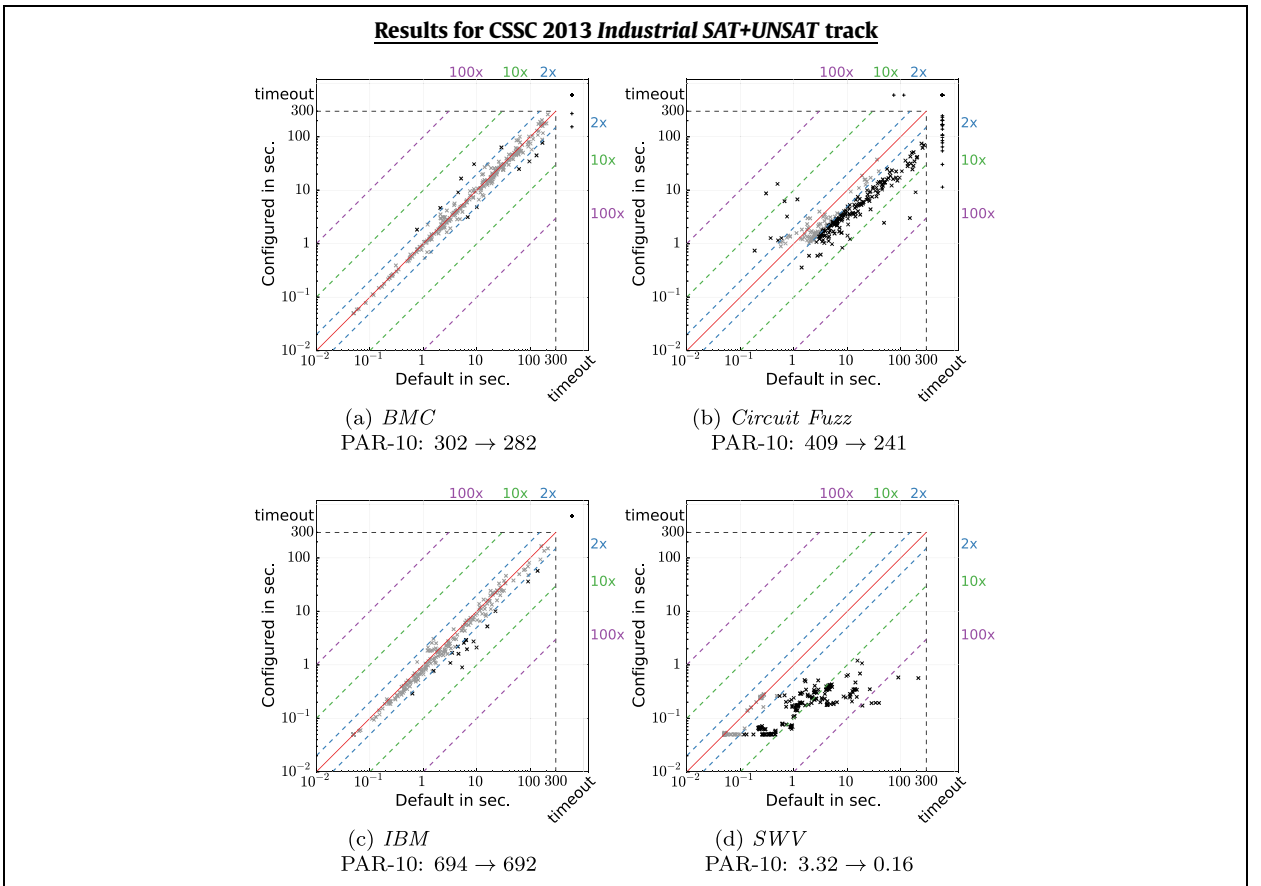


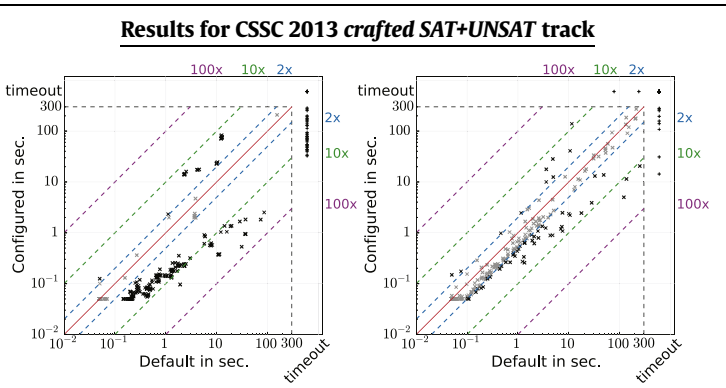
Fig. 1. Speedups achieved by configuration of *Lingeling*. For each benchmark, we show scatter plots of solver defaults vs. configured parameter settings.

Table 4
Results for CSSC 2013 competition track *Industrial SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter setting, bold-facing the better one; we broke ties by the solver's average runtime (not shown for brevity). We aggregated results across all benchmarks to compute the final ranking.

	#timeouts default → #timeouts configured (on test set)					Rank	
	BMC	*Circuit Fuzz*	*IBM*	*SWV*	*Overall*	def	cssc
Lingeling	28 → **26**	39 → **20**	69 → 69	0 → 0	136 → **115**	4	1
Riss3g	32 → **30**	20 → **18**	70 → **69**	0 → 0	122 → **117**	1	2
Solver43	30 → 30	20 → 20	77 → 77	0 → 0	127 → 127	2	3
Forl-nodrup	50 → **36**	33 → **23**	69 → 69	0 → 0	152 → **128**	5	4
Simpsat	38 → **35**	26 → **24**	70 → **69**	0 → 0	134 → **128**	3	5
Clasp-3.0.4-p8	66 → **42**	26 → **17**	71 → 71	0 → 0	163 → **130**	6	6
Sat4j	70 → 70	36 → **30**	77 → **76**	1 → **0**	184 → **176**	7	7
Gnovelty+GCwa	291 → **285**	301 → **295**	295 → 295	244 → **215**	1131 → **1090**	10	8
Gnovelty+PCL	289 → **288**	302 → 302	295 → **294**	215 → 215	1101 → **1099**	8	9
Gnovelty+GCa	291 → **290**	**300** → 302	295 → 295	243 → **217**	1129 → **1104**	9	10

Fig. 1 visualizes the results of the configuration process for the winning solver *Lingeling* on these four benchmark sets. It demonstrates that even *Lingeling*, a highly competitive solver in terms of default performance, can be configured for improved performance on a wide range of benchmarks. We note that for the easy benchmark *SWV*, configuration sped up *Lingeling* by a factor of 20 (average runtime 3.3 s *vs* 0.16 s), and that for the harder *Circuit Fuzz* instances, it nearly halved the number of timeouts (39 *vs* 20). The improvements were smaller for more traditional hardware verification instances (*IBM* and *BMC*) similar to those used to determine *Lingeling*'s default parameter settings.

Table 4 summarizes the results of the ten solvers that were eligible for medals. From this table, we note that, like *Lingeling*, many other solvers benefited from configuration. Indeed, some solvers (in particular *Forl-nodrup* and *Clasp-3.0.4-p8*) benefited much more from configuration on the *BMC* instances, largely because their default performance was worse on this benchmark. On the other hand, *Riss3g* featured stronger default performance than *Lingeling* but did not benefit as much from configuration.



(a) *Graph Isomorphism (GI)* PAR-10: 362 → 65

(b) *Low Autocorrelation Binary Sequence (LABS).* PAR-10: 837 → 779

Fig. 2. Speedups achieved by configuration of *Clasp-3.0.4-p8* on the CSSC 2013 *crafted SAT+UNSAT* track. We show scatter plots of default vs. configured versions of *Clasp-3.0.4-p8*.

Table 5
Results for CSSC 2013 competition track *crafted SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter setting, bold-facing the better one. We aggregated results across all benchmarks to compute the final ranking.

	#TOs default → #TOs configured (on test set)			Rank	
	GI	*LABS*	*Overall*	def	cssc
Clasp-3.0.4-p8	42 → **6**	97 → **90**	139 → **96**	2	1
Forl-nodrup	40 → **7**	95 → **91**	135 → **98**	1	2
Lingeling	43 → **10**	105 → **97**	148 → **107**	3	3
Riss3g	51 → **42**	97 → **89**	148 → **131**	4	4
Simpsat	42 → 42	107 → 107	149 → 149	5	5
Solver43	66 → **65**	90 → **87**	156 → **152**	6	6
Sat4j	62 → **57**	110 → **104**	172 → **161**	7	7
Gnovelty+GCwa	180 → 180	195 → **154**	375 → **334**	8	8
Gnovelty+GCa	183 → **180**	240 → **173**	423 → **353**	10	9
Gnovelty+PCL	179 → **178**	199 → **183**	378 → **361**	9	10

Table 4 also aggregates results across the four benchmark families to yield the overall results for the *Industrial SAT+UNSAT* track. These results show that many solvers benefited substantially from configuration, and that some benefited more than others, causing the CSSC ranking to differ substantially from the ranking according to default solver performance; for instance, based on default performance, the overall winning solver, *Lingeling*, would have only ranked fourth.

4.3.2. Results of the crafted SAT+UNSAT track

The *crafted SAT+UNSAT* track consisted of the two crafted benchmarks detailed in Appendix A.2: *Graph Isomorphism (GI)* and *Low Autocorrelation Binary Sequence (LABS)*.

Fig. 2 visualizes the improvements algorithm configuration yielded for the best-performing solver *Clasp-3.0.4-p8* on these benchmarks. Improvements were particularly large on the *GI* instances, where algorithm configuration decreased the number of timeouts from 42 to 6. Table 5 summarizes the results we obtained for all solvers on these benchmarks, showing that configuration also substantially improved the performance of many other solvers. The table also aggregates results across both benchmark families to yield overall results for the *crafted SAT+UNSAT* track. While *Forl-nodrup* showed the best default performance and benefited substantially from configuration (#timeouts reduced from 135 to 98), *Clasp-3.0.4-p8* improved even more (#timeouts reduced from 139 to 96).

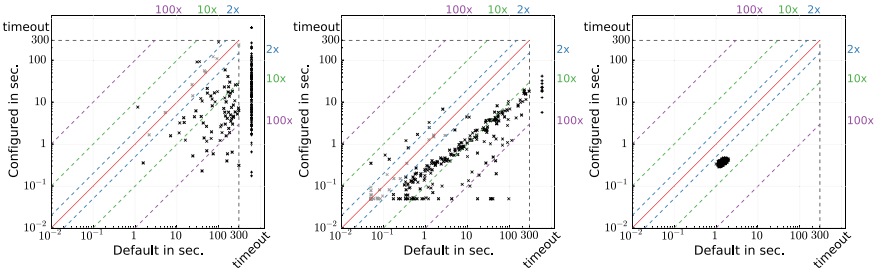
4.3.3. Results of the Random SAT+UNSAT track

The *Random SAT+UNSAT* track consisted of three random benchmarks detailed in Appendix A.3: *5sat500*, *K3*, and *unif-k5*. The instances in *5sat500* were all satisfiable, those in *unif-k5* all unsatisfiable, and those in *K3* were mixed.

Table 6 summarizes the results for these benchmarks. It shows that the *unif-k5* benchmark set was very easy for complete solvers (although configuration still yielded up to 4-fold speedups), that the *K3* benchmark was also quite easy for the best solvers, and that only the SLS solvers could tackle benchmark *5sat500*, with configuration making a big difference to performance.

Here again, our aggregate results demonstrate that rankings were substantially different between the default and configured versions of the solvers: the three solvers with top default performance were ranked 4th to 6th in the CSSC, and vice

Results for CSSC 2013 *Random SAT+UNSAT* track



(a) *Gnovelty+GCa* on *5sat500*
PAR-10: 1997 → 77

(b) *Clasp-3.0.4-p8* on *K3*
PAR-10: 158 → 2.79

(c) *Clasp-3.0.4-p8* on *unif-k5*
PAR-10: 1.44 → 0.37

Fig. 3. Speedups achieved by configuration on the CSSC 2013 *Random SAT+UNSAT* track. We show scatter plots of default vs. configured solvers.

Table 6
Results for CSSC 2013 competition track *Random SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter setting, bold-facing the better one. Results were aggregated across all benchmarks to compute the final ranking. We broke ties by the solver's average runtime. While we do not show runtimes for brevity, the runtimes important for the ranking were the average runtimes of the top 3 solvers on the union of *K3* and *unif-k5*: 1.58 s (*Clasp-3.0.4-p8*), 4.20 s (*Lingeling*), and 7.68 s (*Riss3g*).

	#TOs default → #TOs configured (on test set)				Rank	
	5sat500	*K3*	*unif-k5*	*Overall*	def	cssc
Clasp-3.0.4-p8	250 → 250	11 → **0**	0 → 0	261 → **250**	6	1
Lingeling	250 → 250	8 → **0**	0 → 0	258 → **250**	4	2
Riss3g	250 → 250	10 → **0**	0 → 0	260 → **250**	5	3
Solver43	250 → 250	6 → **3**	0 → 0	256 → **253**	2	4
Simpsat	250 → 250	4 → 4	0 → 0	254 → 254	1	5
Sat4j	250 → 250	7 → **5**	0 → 0	257 → **255**	3	6
Forl-nodrup	250 → 250	39 → **8**	0 → 0	289 → **258**	7	7
Gnovelty+GCwa	8 → **1**	124 → 124	250 → 250	382 → **375**	8	8
Gnovelty+GCa	163 → **4**	124 → 124	250 → 250	537 → **378**	9	9
Gnovelty+PCL	250 → **11**	124 → 124	250 → 250	624 → **385**	10	10

versa. Fig. 3 visualizes the very substantial speedups achieved by configuration for the winning solver *Clasp-3.0.4-p8* on *K3* and *unif-k5*, and for the SLS solver *Gnovelty+GCa* on *5sat500*.

5. The Configurable SAT Solver Challenge 2014

The second CSSC[5] was held in 2014. Compared to the inaugural CSSC in 2013, we improved the competition design in several ways:

- We used a different computer cluster,[6] enabling us to run *GGA* as one of the configuration procedures.
- We added a *Random SAT* track to facilitate comparisons of stochastic local search solvers.
- We dropped the (too easy) SWV benchmark family and introduced four new benchmark families, yielding a total of three benchmark families in each of the four tracks, summarized in Table 7 and described in detail in Appendix A.
- We let solver authors decide which tracks their solver should run in.
- For fairness, for each solver, we performed the same number of configuration experiments. (This is in contrast to 2013, where we performed the same number of configuration runs *for every configuration space* of every solver, which lead to a larger combined configuration budget for solvers submitted with multiple configuration spaces).
- We kept track of all of the (millions of) solver runs performed during the configuration process and made all information about errors available to solver developers after the competition.

[5] http://aclib.net/cssc2014/.

[6] We executed this competition on the META cluster at the University of Freiburg, whose compute nodes contained 64 GB of RAM and two 2.60 GHz Intel Xeon E5-2650v2 8-core CPUs with 20 MB L2 cache each, running Ubuntu 14.04 LTS, 64 bit.

Table 7
Overview of benchmark sets used in the CSSC 2014 tracks *Industrial SAT+UNSAT*, *crafted SAT+UNSAT*, *Random SAT+UNSAT*, and *Random SAT* (from top to bottom); k and m stand for factors of one thousand and one million, respectively.

Benchmark	#Train	#Test	#Variables	#Clauses	Reference
IBM	383	302	96.4k ± 170k	413k ± 717k	[85]
Circuit Fuzz	299	302	5.53k ± 7.45k	18.8k ± 25.3k	[23]
BMC	604	302	424k ± 843k	1.03m ± 2.30m	[18]
GI	1032	351	11.2k ± 17.8k	2.98m ± 8.03m	[68,82]
LABS	350	351	75.9k ± 75.7k	154k ± 153k	[69]
N-Rooks	484	351	38.2k ± 37.4k	125k ± 126k	[67]
K3	300	250	262 ± 43	1116 ± 182	[11]
3cnf	500	250	350 ± 0	1493 ± 0	[12]
unif-k5	300	250	50 ± 0	1056 ± 0	–
3sat1k	250	250	500 ± 0	10000 ± 0	[81]
5sat500	250	250	1000 ± 0	4260 ± 0	[81]
7sat90	250	250	90 ± 0	7650 ± 0	[81]

Table 8
Overview of solvers in the CSSC 2014 and their parameters of various types ('c' for categorical, 'i' for integer, 'r' for real-valued'); 'cond' identifies how many of these parameters are conditional. For each solver, we also list the sizes of the original configuration space submitted by solver developers and of a discretized version, as well as the categories in which the solver participated. Solvers are ordered by the number of parameters they expose ($c + i + r$).

Solver	# Parameters				# Configurations		Categories	Ref.
	c	i	r	Cond.	Discretized	Original		
DCCASat+march-rw	1	0	0	0	9	9	Random	[60]
CSCCSat2014	3	0	0	0	567	567	Random SAT	[59,60]
ProbSAT	5	1	3	4	1×10^5	∞	Random SAT	[9]
Minisat-HACK-999ED	10	0	0	3	8×10^5	8×10^5	All categories	[71]
YalSAT	16	10	0	0	5×10^6	2×10^{72}	Crafted&Random SAT	[17]
Cryptominisat	14	15	7	2	3×10^{24}	∞	Industrial & Crafted	[77]
Clasp-3.0.4-p8	38	30	7	55	1×10^{49}	∞	All categories	[33]
Riss-4.27	214	0	0	160	5×10^{86}	5×10^{86}	All but Random SAT	[64]
SparrowToRiss	170	36	16	176	1×10^{112}	∞	All categories	[8]
Lingeling	137	186	0	0	1×10^{53}	$2 \cdot 10^{1341}$	All categories	[17]

5.1. Participating solvers

The ten solvers that participated in the CSSC 2014 are summarized in Table 8; they included CDCL, SLS and hybrid solvers. These solvers differed substantially in their degree of parameterization, with the number of parameters ranging from 1 to 323. We briefly discuss the main features of each solver's parameter configuration space, ordering solvers by their number of parameters.

DCCASat+march-rw [60] combines the SLS solver DCCASat with the CDCL solver march-rw. It was submitted to the *Random SAT+UNSAT* track. Its only (continuous) parameter is the time ratio of the SLS solver. This parameter was pre-discretized to nine values.

CSCCSat2014 [59,60] is an SLS solver based on configuration checking and dynamic local search methods. It was submitted to the *Random SAT* track. It features 3 continuous parameters that were pre-discretized to 7, 9, and 9 values each, giving rise to a total configuration space of 567 possible parameter configurations. The parameters control the weighting of the dynamic local search part and the probabilities for the linear make functions used in the random walk steps.

ProbSAT [9] is a simple SLS solver based on probability distributions that are built from simple features, such as the make and break of variables [9]. *ProbSAT*'s 9 parameters control the type and the parameters of the probability distribution, as well as the type of restart. *ProbSAT* was submitted to the *Random SAT* track.

Minisat-HACK-999ED [71] is a CDCL solver; it was submitted to all tracks. It has one categorical parameter (whether or not to use the Luby restarting strategy) and 9 numerical parameters fine-tuning the Luby and geometric restart strategies, as well as controlling clause removal and the treatment of glue clauses. 3 of these 9 numerical parameters are conditional on the choice of the Luby restart strategy, and all numerical parameters were pre-discretized by the solver developer. There are also 3 forbidden parameter combinations derived from a weak inequality constraint between two parameter values.

YalSAT [17] is an SLS solver; it was submitted to the tracks *crafted SAT+UNSAT* and *Random SAT*. It has 27 parameters that parameterize the solver's restart component (7 parameters) amongst many other components. 11 of the 27 parameters are numerical, with 6 of them having a trivial upper bound of max-integer ($2^{31} - 1$).

Cryptominisat [77] is a CDCL solver; it was submitted to the tracks *Industrial SAT+UNSAT* and *crafted SAT+UNSAT*. It has 29 parameters that control restarts (6 mostly numerical parameters), clause removal (7 mostly numerical parameters), variable branching and polarity (3 parameters each), simplification (5 parameters), and several other mechanisms. 2 of the numerical parameters further parameterize the blocking restart mechanism and are thus conditional on that mechanism being selected.

Clasp-3.0.4-p8 [33] is a solver for the more general answer set programming (ASP) problem, but it can also solve SAT, MAXSAT and PB problems. It is fundamentally similar to the solver submitted in 2013; changes in the new version focused on the ASP solving part rather than the SAT solving part. As a SAT solver, *Clasp-3.0.4-p8* has 75 parameters, of which 7 control preprocessing, 14 variable selection, 19 the restart policy, 28 the deletion policy and 7 miscellaneous other mechanisms. The configuration space is highly conditional, with several top-level parameters enabling or disabling certain strategies. Finally, there are also 2 forbidden parameter combinations that prevent certain combinations of deletion strategies. *Clasp-3.0.4-p8* exposes both a mixed continuous/discrete parameter configuration space and a manually-discretized one. It was submitted to all tracks.

Riss-4.27 [64] is a CDCL solver submitted to all tracks except *Random SAT*. Compared to the 2013 version *Riss3g*, it almost doubled its number of parameters, yielding 214 parameters organized into 121 simplification and 93 search parameters. In particular, it added many new preprocessing and inprocessing techniques, including XOR handling (via Gaussian elimination [37]), and extracting cardinality constraints [20]. Roughly half of the simplification parameters and a third of the search parameters are categorical (in both cases most of the categoricals are binary). The simplification parameters comprise about 20 Boolean switches for preprocessing techniques and about 100 in-processor parameters, prominently including blocked clause elimination, bounded variable addition, equivalence elimination [34], numerical limits, probing, symmetry breaking, unhiding [39], Gaussian elimination, covered literal elimination [66], and even some stochastic local search. The search parameters parameterize a wide range of mechanisms including variable selection, clause learning and removal, restarts, clause minimization, restricted extended resolution, and interleaved clause strengthening.

SparrowToRiss [8] combines the SLS solver *Sparrow* with the CDCL solver *Riss-4.27* by first running *Sparrow*, followed by *Riss-4.27*. It was submitted to all tracks. *SparrowToRiss*'s configuration space is that of *Riss-4.27* plus 6 *Sparrow* parameters and 2 parameters controlling when to switch from *Sparrow* to *Riss-4.27*: the maximal number of flips for *Sparrow* (by default 500 million) and the CPU time for *Sparrow* (by default 150 seconds). Also, in contrast to *Riss-4.27*, *SparrowToRiss* does not pre-discretize its numerical parameters, but expresses them as 36 integer and 16 continuous parameters.

Lingeling [17] is a successor to the 2013 version; it was submitted to the tracks *Industrial SAT+UNSAT* and *crafted SAT+UNSAT*. Compared to 2013, *Lingeling*'s parameter space grew by roughly a third, to a total of 323 parameters (meaning that again, *Lingeling* was the solver with the most parameters). As in 2013, roughly 40% of these parameters were categorical and the rest integer-valued (many with a trivial upper bound of max-integer, $2^{31} - 1$). Notable groups of parameters that were introduced in the 2014 version include additional preprocessing/inprocessing options and new restart strategies.

5.2. Configuration pipeline

In the CSSC 2014, we used the configurators *ParamILS*, *GGA*, and *SMAC*. For each benchmark and solver, we ran *GGA* and *SMAC* on the solver's full configuration space, which could contain an arbitrary combination of numerical and categorical parameters. We also ran all configurators on a discretized version of the configuration space (automatically constructed unless provided by the solver authors), yielding a total of five configuration approaches: *ParamILS*-discretized, *GGA*, *GGA*-discretized, *SMAC*, and *SMAC*-discretized. *GGA* could not handle the complex conditionals of some solvers; therefore, for these solvers we only ran *ParamILS* and the two *SMAC* variants.

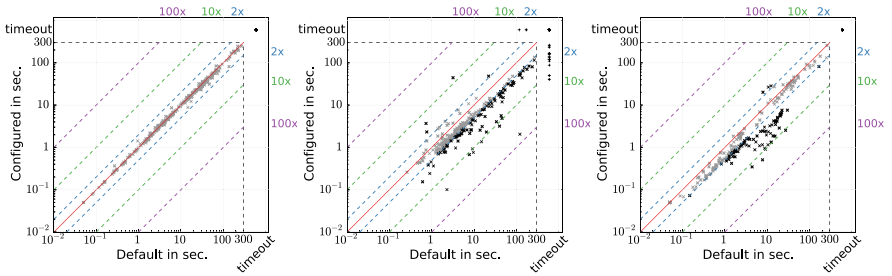
Due to the cost of running a third configurator on nearly every configuration scenario, we reduced the budget for each configuration approach from two CPU days on five cores in CSCC 2013 to two CPU days on four cores in CSSC 2014. In the case of *ParamILS* and *SMAC*, as in 2013, we used these four cores to perform four independent 2-day configurator runs. In the case of *GGA*, we performed one 2-day run using all four cores. We evaluated the configurations resulting from each of the 14 configuration runs (4 *ParamILS*-discretized, 4 *SMAC*-discretized, 4 *SMAC*, 1 *GGA*-discretized, and 1 *GGA*) on the entire training data set of the benchmark at hand and selected the configuration with the best performance. We then executed only this configuration on the benchmark's test set to determine the performance of the configured solver.

In the four tracks of the CSSC (*Industrial SAT+UNSAT*, *crafted SAT+UNSAT*, *Random SAT+UNSAT*, *Random SAT*) we had 6, 6, 5, and 6 participating solvers, respectively, and since there were three benchmark families per track, we ended up with $(6 + 6 + 5 + 6) \times 3 = 72$ pairs of solvers and benchmarks to configure them on. For each of these configuration scenarios, each of the 5 configuration approaches above required four cores for 2 days, yielding a total computational expense of

Table 9
Winners of the four tracks of CSSC 2014.

Rank	*Industrial SAT+UNSAT*	*crafted SAT+UNSAT*	*Random SAT+UNSAT*	*Random SAT*
1st	*Lingeling*	*Clasp-3.0.4-p8*	*Clasp-3.0.4-p8*	*ProbSAT*
2nd	*Minisat-HACK-999ED*	*Lingeling*	*DCCASat+march-rw*	*SparrowToRiss*
3rd	*Clasp-3.0.4-p8*	*Cryptominisat*	*Minisat-HACK-999ED*	*CSCCSat2014*

Results for CSSC 2014 *Industrial SAT+UNSAT* track



(a) *BMC* PAR-10: 222 → 221 (b) *Circuit Fuzz* PAR-10: 316 → 193 (c) *IBM* PAR-10: 697 → 694

Fig. 4. Scatter plots of default vs. configured *Lingeling*, the gold-medal winner of the *Industrial SAT+UNSAT* track of CSSC 2014.

Table 10
Results for CSSC 2014 competition track *Industrial SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter setting, bold-facing the better one. We aggregated results across all benchmarks to compute the final ranking.

	#timeouts default → #timeouts configured (on test set)				Rank	
	BMC	*Circuit Fuzz*	*IBM*	*Overall*	def	cssc
Lingeling	20 → 20	30 → **18**	69 → 69	119 → **107**	2	1
Minisat-HACK-999ED	22 → 22	21 → **19**	70 → 70	113 → **111**	1	2
Clasp-3.0.4-p8	44 → **30**	18 → **12**	71 → 71	133 → **113**	4	3
Riss-4.27	39 → **26**	**20** → 22	72 → 72	131 → **120**	3	4
Cryptominisat	40 → **37**	31 → **20**	70 → **69**	141 → **126**	5	5
SparrowToRiss	62 → **36**	29 → **21**	72 → 72	163 → **129**	6	6

$72 \times 5 \times 4 \times 2 = 2880$ CPU days (close to 8 CPU years). Thanks to a special allocation on the META cluster at the University of Freiburg, we were able to finish this process within 2 weeks.

We note that all scripts we used for performing the configuration and analysis experiments were written in Python (updated from Ruby in 2013) and are available for download on the competition website.

5.3. Results

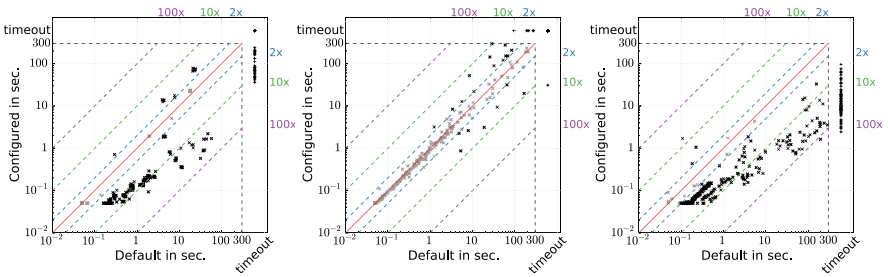
For each of the four tracks of CSSC 2014, we configured the solvers submitted to the track on each of the three benchmark families from that track and aggregated results across the respective test instances. We show the winners for each track in Table 9 and discuss the results in the following sections. Additional details, tables, and figures are provided in an accompanying technical report [48].

5.3.1. Results of the Industrial SAT+UNSAT track

The *Industrial SAT+UNSAT* track consisted of three industrial benchmarks detailed in Appendix A.1: *BMC* [15], *Circuit Fuzz* [23], and *IBM* [85]. Fig. 4 visualizes the results of applying algorithm configuration to the winning solver *Lingeling* on these three benchmark sets. It shows similar results as in the *Industrial SAT+UNSAT* track of CSSC 2013: *Lingeling*'s strong default performance on 'typical' hardware verification benchmarks (IBM and BMC) could only be improved slightly by configuration, but much larger improvements were possible on less standard benchmarks, such as *Circuit Fuzz*.

Table 10 summarizes the results for all six solvers that participated in the *Industrial SAT+UNSAT* track. These results demonstrate that, in contrast to *Lingeling*, several solvers (in particular, *Clasp-3.0.4-p8*, *Riss-4.27*, and *SparrowToRiss*) benefited largely from configuration on the BMC benchmark, but did not reach *Lingeling*'s performance even after configuration. *Minisat-HACK-999ED* performed even better than *Lingeling* with its default parameters, but did not benefit from configuration as much as *Lingeling* (particularly on the *Circuit Fuzz* benchmark family).

Results for CSSC 2014 *crafted SAT+UNSAT* track



(a) *GI* PAR-10: 370 → 90 (b) *LABS* PAR-10: 755 → 804 (c) *N-Rooks* PAR-10: 705 → 5

Fig. 5. Scatter plots of default vs. configured *Clasp-3.0.4-p8*, the gold medal winner of the *crafted SAT+UNSAT* track of CSSC 2014.

Table 11
Results for CSSC 2014 competition track *crafted SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter settings, bold-facing the better one. We aggregated results across all benchmarks to compute the final ranking. *SparrowToRiss* was disqualified from this track, since it returned 'satisfiable' for one instance without producing a model.

	#timeouts default → #timeouts configured (on test set)				Rank	
	GI	*LABS*	*N-Rooks*	*Overall*	def	cssc
Clasp-3.0.4-p8	43 → **9**	**87** → 93	81 → **0**	211 → **102**	5	1
Lingeling	11 → **5**	**101** → 104	3 → **0**	115 → **109**	1	2
Cryptominisat	43 → **24**	95 → **89**	2 → **1**	140 → **114**	3	3
Riss-4.27	43 → **30**	91 → **88**	2 → **0**	136 → **118**	2	4
Minisat-HACK-999ED	50 → 50	91 → 91	0 → 0	141 → 141	4	5
YalSAT	186 → 186	218 → **207**	351 → 351	755 → **744**	6	6
SparrowToRiss(disq.)	55 → **42**	98 → **94**	3 → **0**	156 → **136**	–	–

5.3.2. Results of the crafted SAT+UNSAT track

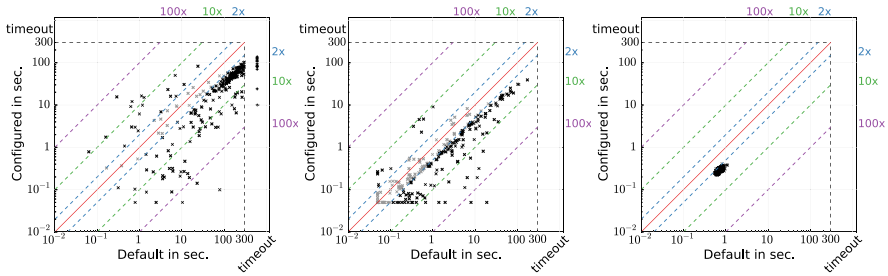
The *crafted SAT+UNSAT* track consisted of the three crafted benchmarks detailed in Appendix A.2: *Graph Isomorphism (GI)*, *Low Autocorrelation Binary Sequence (LABS)*, and *N-Rooks*. Fig. 5 visualizes the improvements configuration yielded on these benchmarks for the best-performing solver, *Clasp-3.0.4-p8*. The effect of configuration was particularly large on the *N-Rooks* instances, where it reduced the number of timeouts from 81 to 0. Similar to the results from CSSC 2013, configuration also substantially improved performance on the *GI* instances, decreasing the number of timeouts from 43 to 9. In contrast to 2013, an unusual effect occurred for *Clasp-3.0.4-p8* on the *LABS* instances, where the number of timeouts on the test set *increased* from 87 to 93 by configuration; we study the reasons for this in more detail in Section 6.1.

Table 11 summarizes the results of all solvers on the *crafted SAT+UNSAT* track, showing that the performance of many other solvers also substantially improved on the benchmarks *GI* and *N-Rooks*, and only mildly (if at all) on the *LABS* benchmark. The aggregate results across these 3 benchmark families show that *Lingeling* had the best default performance, but only benefited mildly from configuration (#timeouts reduced from 115 to 109), whereas *Clasp-3.0.4-p8* benefited much more from configuration and thus outperformed *Lingeling* after configuration (#timeouts reduced from 211 to 102). Once again, we note that the winning solver only showed mediocre performance based on its default: *Clasp-3.0.4-p8* would have ranked 5th in a comparison based on default performance.

5.3.3. Results of the Random SAT+UNSAT track

The *Random SAT+UNSAT* track consisted of three random benchmarks detailed in Appendix A.3: *3cnf*, *K3*, and *unif-k5*. The instances in *unif-k5* are all unsatisfiable, while the other two sets contain both satisfiable and unsatisfiable instances. Fig. 6 visualizes the improvements achieved by configuration on these benchmarks for the best-performing solver *Clasp-3.0.4-p8*. *Clasp-3.0.4-p8* benefited most from configuration on benchmark *3cnf*, where it reduced the number of timeouts from 18 to 0. For the other benchmarks, it could already solve all instances in its default parameter configuration, but configuration helped reduce its average runtime by factors of 3 (*K3*) and 2 (*unif-k5*), respectively. Table 12 summarizes the results of all solvers for these benchmarks. We note that solver *DCCASat+march-rw* showed the best default performance, and that after configuration, it also solved all instances from the three benchmark sets, only ranking behind *Clasp-3.0.4-p8* because the latter solved these instances faster.

Results for CSSC 2014 *Random SAT+UNSAT* track



(a) *3cnf* PAR-10: 309 → 35 (b) *K3* PAR-10: 7.91 → 2.66 (c) *unif-k5* PAR-10: 0.74 → 0.30

Fig. 6. Scatter plots of default vs. configured *Clasp-3.0.4-p8*, the gold medal winner of the *Random SAT+UNSAT* track of CSSC 2014.

Table 12
Results for CSSC 2014 competition track *Random SAT+UNSAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter settings, bold-facing the better one; we broke ties by the solver's average runtime (not shown for brevity, but the average runtimes important for tie breaking were 13 seconds for configured *Clasp-3.0.4-p8* and 21 seconds for configured *DCCASat+march-rw*). We aggregated results across all benchmarks to compute the final ranking.

	#timeouts default → #timeouts configured (on test set)				Rank	
	3cnf	*K3*	*unif-k5*	*Overall*	def	cssc
Clasp-3.0.4-p8	18 → **0**	0 → 0	0 → 0	18 → **0**	2	1
DCCASat+march-rw	1 → **0**	0 → 0	1 → **0**	2 → **0**	1	2
Minisat-HACK-999ED	166 → **99**	5 → **1**	0 → 0	171 → **100**	5	3
Riss-4.27	160 → **113**	2 → 2	1 → **0**	163 → **115**	4	4
SparrowToRiss	126 → 126	8 → **1**	0 → 0	134 → **127**	3	5

5.3.4. Results of the Random SAT track

The *Random SAT* track consisted of the three benchmarks detailed in Appendix A.3: *3sat1k*, *5sat500* and *7sat90*. Fig. 7 visualizes the improvements configuration achieved on these benchmarks for the best-performing solver *ProbSAT*. *ProbSAT* benefited most from configuration on benchmark *5sat500*: its default did not solve a single instance in the maximum runtime of 300 seconds, while its configured version solved all instances in an average runtime below 2 seconds! Since timeouts at 300 s yield a PAR-10 score of 3000, the PAR-10 speedup factor on this benchmark was 1 500, the largest we observed in the CSSC. On the other two scenarios, configuration was also very beneficial, reducing *ProbSAT*'s number of timeouts from 24 to 0 (7sat90) and from 10 to 4 (3sat1k), respectively. Table 13 summarizes the results of all solvers for these benchmarks, showing that next to *ProbSAT*, only *SparrowToRiss* benefited from configuration. Neither of the CDCL solvers (*Clasp-3.0.4-p8* and *Minisat-HACK-999ED*) solved a single instance in any of the three benchmarks (in either default or configured variants). For the other two SLS solvers, *YalSAT* and *CSCCSat2014*, the defaults were already well tuned for these benchmark sets. Indeed, we observed overtuning to the training sets in one case each: *YalSAT* for *3sat1k* and *CSCCSat2014* for *7sat90*. Overall, the configurability of *ProbSAT* and *SparrowToRiss* allowed them to place first and second, respectively, despite their poor default performance (especially on *5sat500*, where neither of them solved a single instance with default settings).

6. Post-competition analyses

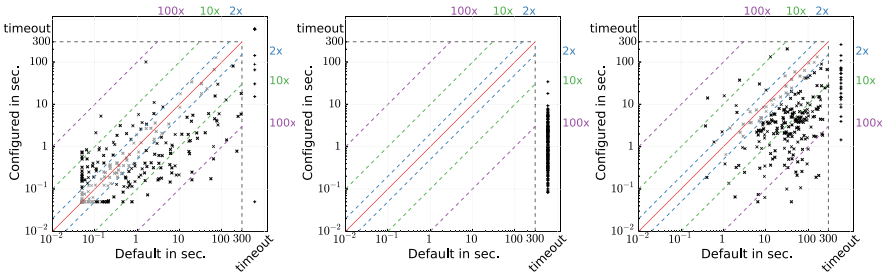
While the previous sections focused on the results of the respective competitions, we now discuss several analyses we performed afterwards to study overarching phenomena and general patterns.

6.1. Why does configuration work so well and how can it fail?

Several practitioners have asked us why automated configuration can yield the large speedups over the default configuration we observed. We believe there are two key reasons for this:

- No single algorithmic approach performs best on all types of benchmark instances; this is precisely the same reason that algorithm selection approaches (such as SATzilla [84] or 3S [52]) work so well.
- Solver defaults are typically chosen to be robust across benchmark families. For any given benchmark family F, highly parameterized solvers can, however, typically be instantiated to exploit the idiosyncrasies of F substantially better. (These improvements only need to generalize to other instances from F, not to other benchmark families.)

Results for CSSC 2014 *Random SAT* track



(a) *3sat1k* PAR-10: 132 → 53 (b) *5sat500* PAR-10: 3000 → 2 (c) *7sat90* PAR-10: 337 → 15

Fig. 7. Scatter plots of default vs. configured *ProbSAT*, the gold medal winner of the *Random SAT* track of CSSC 2014.

Table 13
Results for CSSC 2014 competition track *Random SAT*. For each solver and benchmark, we show the number of test set timeouts achieved with the default and the configured parameter settings, bold-facing the better one; we broke ties by the solver's average runtime (not shown for brevity). We aggregated results across all benchmarks to compute the final ranking.

	#timeouts default → #timeouts configured (on test set)				Rank	
	3sat1k	*5sat500*	*7sat90*	*Overall*	def	cssc
ProbSAT	10 → **4**	250 → **0**	24 → **0**	284 → **4**	4	1
SparrowToRiss	9 → **5**	250 → **0**	3 → 3	262 → **8**	3	2
CSCCSat2014	2 → 2	0 → 0	**3** → 6	**5** → 8	1	3
YalSAT	**6** → 7	0 → 0	5 → 5	**11** → 12	2	4
Clasp-3.0.4-p8	250 → 250	250 → 250	250 → 250	750 → 750	5	5
Minisat-HACK-999ED	250 → 250	250 → 250	250 → 250	750 → 750	6	6

However, algorithm configuration does not necessarily work in all cases. For example, in the *crafted SAT+UNSAT* track of the CSSC 2014, we encountered a case in which the configured solver performed somewhat *worse* than the default solver: *Clasp* configured on benchmark family *LABS* timed out on 93 test instances, whereas its default only timed out on 87 test instances (see also Fig. 5b). Two obvious causes suggest themselves in the case of such a failure:

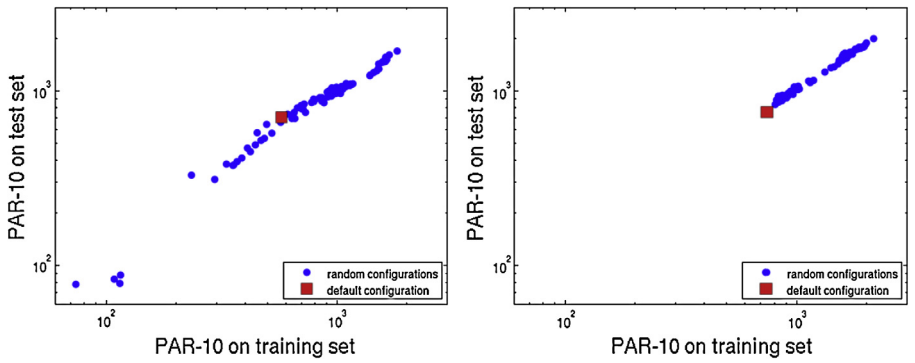
- insufficiently long configuration runs (which can result in worse-than-default performance on the *training* set[7]); and/or
- overtuning on the training set that does not generalize to the *test* set.

We investigated the configuration of *Clasp* on *LABS* further after the competition, and found that in this case *both* of these effects applied: In the CSSC, training performance slightly deteriorated (86 → 87 timeouts); and the improved training performance we found with a larger configuration budget[8] afterwards (86 → 83 timeouts) also did not generalize to the test set (87 → 88 timeouts).

To contrast the conditions under which configuration can fail and under which it works well, we compared the configuration of *Clasp* on benchmarks *LABS* (87 → 93 test timeouts in the CSSC) and *N-Rooks* (81 → 0 test timeouts in the CSSC). For this analysis, we sampled 100 *Clasp* configurations uniformly at random and evaluated their PAR-10 training and test scores on both of the benchmarks; Fig. 8 shows the result. The first observation we make directly based on the figure is that for *N-Rooks*, about 20% of the random configurations outperform the default, whereas for *LABS* none of them do: the default is simply very good to start with for *LABS* and thus much harder to beat. Second, since several configurations are very good (i.e., fast) for *N-Rooks*, configurators can make progress much faster (and also take full advantage of adaptive capping to limit the time spent with poor configurations); indeed, in the CSSC the configurators managed to perform about 8 times more *Clasp* runs in the same time (2 days) for *N-Rooks* than for *LABS* (averaging about 14 400 vs. 1 800 runs). This explains why configurators require more time to improve training performance on *LABS*. Third, to assess the potential for overtuning, we studied how training and test performance correlate. Visually, Fig. 8 shows a strong overall correlation of PAR-10 training and test scores for both of the benchmarks; Spearman correlation coefficients are indeed high: 0.99 (*N-Rooks*) and 0.98 (*LABS*). However, for the top 20% of sampled configurations, the correlation is much stronger for *N-Rooks* (0.98) than for *LABS* (0.49). This explains why improvements on the *LABS* training set do not necessarily translate to improvements on its test set.

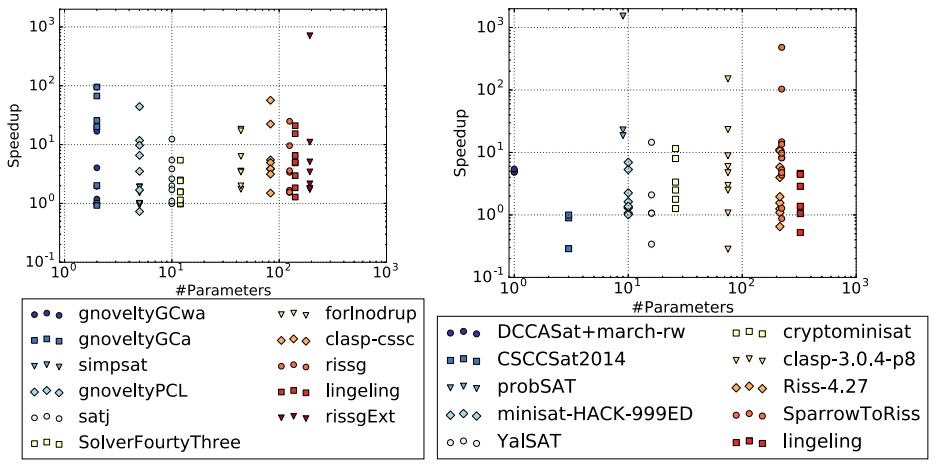
[7] Worse-than-default performance on the training set is possible since configurators only base their decisions on a *subset* of the training instances; that subset increases over time and only reaches the full training set when the configuration process is given enough time.

[8] Specifically, we ran 32 *SMAC* runs for 10 days each.



(a) *N-Rooks* (b) *LABS*

Fig. 8. Training *vs* test PAR-10 scores of 100 random *Clasp* configurations and the *Clasp-3.0.4-p8* default.



(a) CSSC 2013 (b) CSSC 2014

Fig. 9. Number of solver parameters vs. PAR-10 speedup factor of configured over default solver. The speedup factor for each solver considers only instances that were solved by at least one of the default solver and the configured solver. Each symbol denotes one benchmark the solver was run on. Fig. D.11 in the appendix shows the same figure based on PAR-1 for comparison.

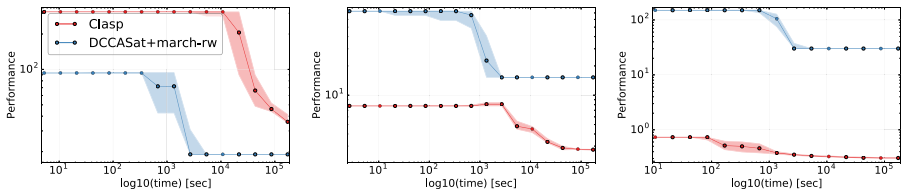
6.2. Overall configurability of solvers

Some solvers consistently benefited more from configuration than others. Here, we quantify the *configurability* of a solver on a given benchmark by the PAR-10 speedup factor its configured version achieved over its default version, computed on the set of instances solved by at least one of the two. We then examine the relationship between configurability and number of parameters to determine whether solvers with many parameters consistently benefited more or less from configuration than solvers with few parameters.[9]

Fig. 9 shows that configurability was indeed high for solvers with many parameters (e.g., the variants of *Lingeling*, *Riss*, and *Clasp*), but that it did not increase monotonically in the number of parameters: some solvers with very few parameters were surprisingly configurable. For example, configuration sped up the single-parameter solver *DCCASat+march-rw* by at least a factor of four in all three benchmarks it was configured for, while the 4-parameter solver *CSCCSat2014* was not improved at all by configuration. Furthermore, *ProbSAT*, which achieved the best single-benchmark performance improvement (as previously discussed in Section 5.3.4), has only 9 parameters.

We note that the notion of configurability used here is strongly dependent on the time budget available for configuration. In the next section, we investigate this issue in more detail.

[9] Of course, it is simple to construct examples where a solver with a single parameter is highly configurable (*e.g.*, let the parameter have a poor default setting) or where a solver has many parameters but does not benefit from configuration at all (*e.g.*, a solver could expose many parameters that are not actually used at all). The focus of our analysis is therefore on the relationship between configurability and the number of parameters that a solver author reasonably expected would be useful to expose.



(a) *3cnf* (b) *K3* (c) *unif-k5*

Fig. 10. Performance on the CSSC 2014 track *Random SAT+UNSAT* achieved by configuration of *DCCASat+march-rw* (1 parameter) and *Clasp-3.0.4-p8* (75 parameters) as a function of the configuration budget. For each of the solvers, we plot mean ± one standard deviation of the PAR10 performance of the incumbent configurations found by 4 runs of *SMAC* over time.

6.3. Impact of configuration budget

The runtime budget we allow to configure each solver has an obvious impact on the results. In one extreme case, if we let this budget go towards zero, the configuration pipeline returns the solver defaults (and we are back in the setting of the standard SAT competition). For small, non-zero budgets, we can expect solvers with few parameters to benefit from configuration more, since their configuration spaces are easier to search. On the other hand, if we increase the time budget, solvers with larger parameter spaces are likely to benefit more than those with smaller parameter spaces (since larger parts of their configuration space can be searched given additional time).

Fig. 10 illustrates this phenomenon for the two top solvers in the *Random SAT+UNSAT* track of CSSC 2014. With the competition's configuration budget of two days across 4 cores, *Clasp-3.0.4-p8* performed better than *DCCASat+march-rw* (both solved all test instances, with average runtimes of 13 vs. 21 seconds). In the extreme case of no time budget for configuration, *DCCASat+march-rw* would have won against *Clasp-3.0.4-p8*, since its default version performed much better (2 vs. 18 timeouts), and, in fact, Fig. 10a shows that it required a configuration budget of at least 10^4 seconds to find improving *Clasp-3.0.4-p8* parameters for the *3cnf* benchmark (where the default version of *Clasp-3.0.4-p8* produced 18 timeouts). While the configuration of *DCCASat+march-rw*'s single parameter had long converged by 10^4 seconds, the configuration of *Clasp-3.0.4-p8*'s 75 parameters continued to improve performance until the end of the configuration budget, and, in particular for the *3cnf* benchmark, performance would have likely continued to improve further if the budget had been larger.

We thus conclude that the solver's flexibility should be chosen in relation to the available budget for configuration: solvers with few parameters can often be improved more quickly than highly flexible solving frameworks, but, given enough computational resources and powerful configurators, the latter ones can typically offer a greater performance potential.

6.4. Results with an increased cutoff time for validation

Next to the overall time budget allowed for configuration, another important time limit is the cutoff time allowed for each single solver run; due to our limited overall budget, we chose this to be quite low: 300 seconds both for solver runs during the configuration process and for the final evaluation of solvers on previously unseen test instances.

Here, we study how using a larger cutoff time at evaluation time affects results, mimicking a situation where we care about performance with a large cutoff time but use a smaller cutoff time for the configuration process to make progress faster. In fact, several studies in the literature (e.g., [42,55,81]) used a smaller cutoff time for configuration than for testing, and we found that improvements with a time budget around 300 seconds often lead to improvements with larger cutoff times.

Table 14 shows the results we obtained when using a cutoff time of 5000 seconds for validation (the same as the SAT competition) for the *Industrial SAT+UNSAT* track of CSSC 2014. Qualitatively, these results are quite similar to those obtained with an evaluation cutoff time of 300 s (compare Table 10), with only few differences. As expected, given the larger cutoff time, all solvers solved substantially more instances (especially for the *BMC* and *Circuit Fuzz* benchmarks). Nevertheless, with a cutoff time of 5000 seconds, for all solvers, the configured variant (configured to perform well with a cutoff time of 300 seconds) still performed better than the default version, making us more confident that configuration does not substantially overtune to achieve good performance on easy instances only.

6.5. Results with a single configurator

While the CSSC addressed the performance of SAT solvers rather than the performance of configurators, we have been asked whether our complex configuration pipeline was necessary, or whether a single configurator would have produced similar or identical results. Indeed, counting the choice of discretized *vs* non-discretized parameter space, our pipeline used five configuration approaches (*ParamILS*-discretized, *GGA*, *GGA*-discretized, *SMAC*-discretized, and *SMAC*). Thus, if one of these approaches had yielded the same results all by itself, we could have reduced our overall configuration budget five-fold.

To determine whether this was the case, we evaluated the solver performance we would have observed if we had used each configuration approach in isolation. For each configuration scenario and each approach, we computed the *PAR-10*

Table 14
Results for CSSC 2014 competition track *Industrial SAT+UNSAT*, when final solvers are evaluated with a per-run cutoff time of 5000 seconds instead of 300 seconds (as in Table 10). The ranking is the same for the default and configured algorithms, but this can be attributed to chance as there are two ties in terms of timeouts, which are only broken by average runtimes: *Minisat-HACK-999ED* (513.7 s) *vs Riss-4.27* (520.5 s), and *SparrowToRiss* (548.5 s) *vs Cryptominisat* (555.0 s).

	#timeouts default → #timeouts configured (on test set)				Rank	
	BMC	*Circuit Fuzz*	*IBM*	*Overall*	def	cssc
Lingeling	10 → 10	6 → **3**	66 → **65**	82 → **78**	1	1
Minisat-HACK-999ED	**11** → 12	6 → **4**	67 → 67	84 → **83**	2	2
Riss-4.27	19 → **12**	3 → 3	70 → **68**	92 → **83**	3	3
SparrowToRiss	19 → **14**	3 → 3	70 → **69**	92 → **86**	4	4
Cryptominisat	20 → **18**	9 → **4**	66 → **64**	95 → **86**	5	5
Clasp-3.0.4-p8	31 → **18**	3 → 3	70 → **69**	104 → **90**	6	6

Table 15
Geometric mean of *PAR-10 slowdown factors over the CSSC result* (i.e., over using the oracle best of our five configuration approaches), computed across the CSSC 2014 scenarios for which the respective solver was submitted. Note that on the training set 1.0 is a lower bound for this metric, while on the test set values smaller than 1.0 are possible if one configuration generalizes better than another. Entry '–' denotes that *GGA* was incompatible with the configuration space (see Appendix B.2 for details).

	Training					Test				
	SMAC-d	*SMAC-c*	*PILS*	*GGA-d*	*GGA-c*	*SMAC-d*	*SMAC-c*	*PILS*	*GGA-d*	*GGA-c*
DCCASat+march-rw	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
CSCCSat2014	1.0	1.0	1.0	3.0	1.6	1.0	1.0	1.0	1.6	1.0
ProbSAT	1.0	1.2	1.1	–	–	1.0	1.5	1.1	–	–
Minisat-HACK-999ED	1.2	1.1	1.2	1.5	2.0	0.9	1.0	1.0	1.0	1.8
YalSAT	1.2	2.2	2.0	18.1	18.5	1.1	0.6	1.8	6.6	12.3
Cryptominisat	1.1	1.2	1.4	3.4	3.9	1.0	1.3	1.4	3.5	4.1
Clasp-3.0.4-p8	1.1	1.5	1.9	–	–	1.2	0.9	1.5	–	–
Riss-4.27	1.1	1.1	1.4	10.5	10.1	1.2	0.9	1.1	6.0	7.4
SparrowToRiss	1.3	1.1	2.0	–	–	1.1	1.0	1.8	–	–
Lingeling	1.5	1.6	1.4	–	77.2	1.4	1.0	1.3	–	30.5

slowdown factor over the CSSC result as the PAR-10 achieved with the respective approach, divided by the PAR-10 of the approach with best training performance (which we selected in the CSSC). If a configuration approach achieves a PAR-10 slowdown factor close to one, this means that it gives rise to solver performance close to that achieved by our full CSSC configuration pipeline. For each solver, we then computed the geometric mean of these factors across the scenarios it was configured for.

Table 15 shows that both *SMAC* variants performed close to best for all solvers, meaning that we would have achieved similar results had we only used *SMAC* in the CSSC. *ParamILS* yielded the next best performance, followed by *GGA*. Full results can be found in the accompanying technical report [48]. Despite *SMAC*'s strong performance, we believe it will still be useful to run several configuration approaches in future CSSCs, both to ensure robustness and to assess whether some configuration scenarios are better suited to other configuration approaches.

7. Conclusion

In this article, we have described the design of the Configurable SAT Solver Challenge (CSSC) and the details of CSSC 2013 and CSSC 2014. We have highlighted two main insights that we gained from this competition:

1. Automated algorithm configuration often improved performance substantially, in several cases yielding average speedups of orders of magnitude.
2. Some solvers benefited more from automated configuration than others, leading to substantially different algorithm rankings after configuration than before (as, *e.g.*, measured by the SAT competition).

Also, the configuration budget influenced which algorithm would perform best, and with the competition budget of 2 days on 4–5 cores, algorithms with larger parameter spaces exhibited more capacity for improvement.

These conclusions have interesting implications for algorithm design: if an algorithm is likely to be applied across a range of specialized applications, then it should be made flexible by parameterization of its key mechanisms and components, and this flexibility should be exploited by automated algorithm configuration. Our findings thus challenge the traditional approach to solver design that tries to avoid having too many algorithm parameters (since these parameters complicate manual tuning and analysis). Rather, they promote the design paradigm of *Programming by Optimization (PbO)* [41], which aims to avoid premature design choices and to rather actively develop promising alternatives for parts of the design that enable an automated customization to achieve peak performance on particular benchmarks of interest. Indeed, in the CSSC, we have

already observed a trend towards PbO, as evidenced by the introduction of a host of new parameters into state-of-the-art solvers, such as *Riss-4.27* and *Lingeling*, between 2013 and 2014.

Finally, there is no reason why a configurable solver competition should be appropriate and insightful only for SAT. On the contrary, similar events would be interesting in the context of many other challenging computational problems, such as answer set programming, constraint programming or AI planning. Another interesting application domain is automatic machine learning, where algorithm configuration can adapt flexible machine learning frameworks to each new dataset at hand [80,32]. We believe that for those and many other problems, similar findings to those we reported here for CSSC would be obtained, leading to analogous conclusions regarding algorithm design.

Acknowledgements

Many thanks go to Kevin Tierney for his generous help with running *GGA*, including his addition of new features, his suggestion of parameter settings and his conversion script to read the pcs format. We also thank the solver developers for proofreading the description of their solvers and their parameters. For computational resources to run the competition, we thank Compute Canada (CSSC 2013) and the German Research Foundation (DFG; CSSC 2014). F. Hutter and M. Lindauer thank the DFG for funding this research under Emmy Noether grant HU 1900/2-1. H. Hoos acknowledges funding through an NSERC Discovery Grant.

Appendix A. Benchmark sets used

We mirrored the three main categories of instances from the SAT competition: industrial, crafted, and random. In 2014, we also included a category of satisfiable random instances from the SAT Races. For each of these categories, we used various benchmark sets, each of them split into a training set to be used for algorithm configuration and a disjoint test set.

For each category, to weight all benchmarks equally, we used the same number of test instances from each benchmark; these test sets were subsampled uniformly at random from the respective complete test sets.

All benchmarks are summarized in Tables 1 and 7 in the main text.

A.1. Industrial benchmark sets

SWV This set of SAT-encoded software verification instances consists of 604 instances generated with the CALYSTO static checker [4], used for the verification of five programs: the spam filter Dspam, the SAT solver HyperSAT, the Wine Windows OS emulator, the gzip archiver, and a component of xinetd (a secure version of inetd). We used the same training/test split as Hutter et al. [42], containing 302 training instances and 302 test instances. We used this benchmark set in the 2013 CSSC. (In 2014, we only used it for preliminary tests since it is quite easy for modern solvers.)

Hardware Verification (IBM) This set of SAT-encoded bounded model checking instances consists of 765 instances generated by Zarpas [85] These instances were originally selected by Hutter et al. [42] as the instances in 40 randomly-selected folders from the IBM Formal Verification Benchmarks Library. We used their original training/test split, containing 382 training instances and 383 test instances. We used this benchmark set in both the 2013 and 2014 CSSCs.

Circuit Fuzz These instances were produced by a circuit-based CNF fuzzing tool, *FuzzSAT* [23] (version 0.1). As *FuzzSAT* was originally designed to produce semi-realistic test cases for debugging SAT solvers, the majority of the instances it produces are trivial; however, occasionally, it produces more challenging instances. The CircuitFuzz instances were found by generating 10 000 *FuzzSAT* instances and removing all those that could be solved within one second by *Lingeling*. This instance generator was originally described in detail by Bayless et al. [11]; we used the 300 instances from that paper as the training set (except one quite easy instance, 'fuzz_100_25433.cnf', which was dropped unintentionally by a script) and produced 585 additional instances using the same method, to form a testing set. We used this benchmark set in both the 2013 and 2014 CSSCs. We used these instances as part of the industrial track since they are "structured in ways that resemble (at least superficially) real-world, circuit-derived instances" [11]; a case could, however, also be made for them to be part of the crafted or random track.

Bounded Model Checking 2008 (BMC) This set of SAT instances was derived by unrolling the 2008 Hardware Model Checking Competition circuits [18]. Each of these instances is a sequential circuit with safety properties. Each circuit was unrolled to 50, 100, and 200 iterations using the tool *aigunroll* (version 1.9.4) from the AIGER tools [15]. We omitted trivial instances that were proven SAT or UNSAT during the unrolling process. While we used the entire set in 2013, in 2014 we removed the 60 instances provided by Intel in order to allow us to publicly share the instances.

A.2. Crafted benchmark sets

Graph Isomorphism (GI) These instances were first used in the 2013 SAT Competition [68] and were generated by encoding the graph isomorphism problem to SAT according to the procedure described by Torán [82]. Given two graphs G_1 and G_2

with n vertices and m edges (for whom the isomorphism problem is to be solved) the generator creates a SAT formula with n^2 variables and $O(n) + O(n^3) + O(n^4)$ clauses. Consequently, the generated instances can contain very many clauses. The 2 064 SAT instances in this set were generated from different types of graphs, with the number of vertices n ranging from 10 to 1296.[10] We split the instances uniformly at random into 1 032 training and 1 032 test instances; in both the 2013 and 2014 CSSCs, we only used 351 of the test instances.

Low Autocorrelation Binary Sequence (LABS) This set contains 651 low-autocorrelation binary sequence (LABS) search problems that were encoded to SAT problems by first encoding them as pseudo-Boolean problems and then as SAT problems. Instances from this set were first used in the SAT Competition 2013 in the crafted category [69]. We split this benchmark set uniformly at random into 350 training and 351 test instances, and used it in both the 2013 and 2014 CSSCs.

N-Rooks These 835 instances [67] represent a parameterized unsatisfiable variation of the well-known n-queens problem, in which the task is to place n queens on a chess board with $n \times n$ fields such that they do not attack each other. In the variation considered here, the (unsatisfiable) problem is to either place $n+1$ *rooks* or $n+1$ queens on a board of size $n \times n$. Additional constraints enforcing that there is a piece in each row/column/diagonal make it easier to prove unsatisfiability, and these constraints can be enabled or disabled by generator parameters. We used the generator *new-Dame* provided by Norbert Manthey to generate instances with $n \in [10, 50]$, using all rooks or all queens, using six different problem encodings, and using all combinations of enabling/disabling all types of constraints. We then removed trivial instances, ending up with 835 instances. For the CSSC 2014, we selected 484 training instances uniformly at random and used the remaining 351 as test instances.

A.3. Random benchmark sets

K3 This is a set of 600 randomly-generated 3-SAT instances at the phase transition (clause to variable ratio of approximately 4.26). It includes both satisfiable and unsatisfiable instances. The set includes 100 instances each with 200 variables (853 clauses), 225 variables (960 clauses), 250 variables (1066 clauses), 275 variables (1172 clauses), 300 variables (1279 clauses), and 325 variables (1385 clauses). These 600 instances were generated by Lin Xu using the random instance generator from the 2009 SAT competition, and were previously described by Bayless et al. [11]. We employed their uniform random split into 300 training and 300 test instances, using all 300 test instances in the CSSC 2013 (random track) and only a subset of 250 test instances in the CSSC 2014 (random track).

3cnf This is a set of 750 random 3-SAT instances (satisfiable and unsatisfiable) at the phase transition, with 350 variables and 1493 clauses. These instances were generated by the ToughSAT instance generator [12] and split into 500 training and 250 test instances uniformly at random. We used this benchmark set in the 2014 CSSC (random track).

unif-k5 This set contains only unsatisfiable 5-SAT instances generated uniformly at random with 50 variables and 1 056 clauses (a clause-to-variable ratio sharply on the phase transition). The instances were generated by the uniform random generator used in the SAT Challenge 2012 and SAT Competition 2013, with satisfiable instances being filtered out by running the SLS solver *ProbSAT*. We used this benchmark set in both the 2013 and 2014 CSSCs (random track).

3sat1k This is a set of 500 3-SAT instances at the phase transition, all satisfiable. Each instance has 1000 variables and 4260 clauses. These instances were previously described by Tompkins et al. [81]. We used their uniform random split into 250 training and test instances in the 2013 CSSC (random track) and in the 2014 CSSC (random satisfiable track).

5sat500 This set contains 500 5-SAT instances generated uniformly at random with a clause-to-variable ratio of 20. Each instance is satisfiable and has 500 variables and 10000 clauses. This set was first used for tuning the SAT solver Captain Jack and other SLS solvers [81]. We used the original uniform random split into 250 training and test instances in the 2014 CSSC (random satisfiable track).

7sat90 This set contains 500 7-SAT instances generated uniformly at random with a clause-to-variable ratio of 85. Each instance is satisfiable and has 90 variables and 7650 clauses. This set was also first used for tuning the SAT solver Captain Jack and other SLS solvers [81]. We used the original uniform random split into 250 training and test instances in the 2014 CSSC (random satisfiable track).

A.4. Instance features used for these benchmark sets

As described in Appendix B.3, *SMAC* can use instance features to guide its search. Such instance features have predominantly been studied in the work on SATzilla for algorithm selection [70,84] and in machine learning models for predicting

[10] Note that the larger graphs have varying node degrees, and that each node can only match with other nodes of the same degree; this allows the encoding to generate much fewer clauses than in the worst case of equal node degrees.

algorithm runtime [49]. These features range from simple summary statistics, such as the number of variables or clauses in an instance, to the results of short, runtime-limited probes with local search solvers. In the context of algorithm configuration, we can afford somewhat more expensive features than for algorithm selection since we only require them on the *training* instances (not the test instances) and can compute them once, offline. Nevertheless, we kept feature computation costs low to not add substantially to the time required for algorithm configuration.

For the instance sets where we already had available instance features from previous work, we used those features. In particular, we used the 138 features described by Hutter et al. [49] for the datasets *SWV*, *IBM*, *3sat1k*, *5sat500*, and *7sat90*. For the set *unif-k5*, we did not compute features since these instances were very easy to solve even with algorithm defaults (note that *SMAC* also worked very well without features). For the other datasets, we computed a subset of 119 features, including basic features and feature groups based on survey propagation, clause learning, local search probing, and search space size estimates.[11]

Appendix B. Configuration procedures

This appendix describes the configuration procedures we used in more detail. Configurators typically iterate the following steps: (1) execute the target algorithm on one or more instances with one or more configurations for a limited amount of time; (2) measure the resulting performance metric and (3) decide upon the next target algorithm execution. Beyond the key question of which configuration to try next, configurators also need to decide how many runs and which instances to use for each evaluation, and after which time to terminate unsuccessful runs. *ParamILS*, *SMAC*, and *GGA* differ in how they instantiate these components.

B.1. ParamILS: local search in configuration space

ParamILS [47], short for iterated local search in parameter configuration space, generalizes the simple (often manually performed) tuning approach of changing one parameter at a time and keeping changes if performance improves. While that simple tuning approach is a local search that terminates in the first local optimum, *ParamILS* carries out an iterated local search [58] that applies perturbation steps in each local optimum o in order to escape o's basin of attraction and carry out another local search that leads to another local optimum o'. Iterated local search then decides whether to continue from the new optimum o' or to return to the previous optimum o, thereby performing a biased random walk over locally optimal solutions. *ParamILS* only supports categorical parameters, so numerical parameters need to be discretized before *ParamILS* is run.

ParamILS is an algorithm framework with two different instantiations that differ in their strategy of deciding how many runs to use to evaluate each configuration. The most straightforward instantiation, *BasicILS*(N), resembles the approach most frequently used in manual parameter optimization: it evaluates each configuration according to a fixed number of N runs on a fixed set of instances. While this approach is simple and intuitive, it gives rise to the problem of how to set the number N. Setting N to a large value yields slow evaluations; using a small number yields fast evaluations, but the evaluations are often not representative for the instance set Π (for example, if we choose N runs we can cover at most N instances, even if we only allow a single run per instance). The second *ParamILS* instantiation, *FocusedILS*, solves this problem by allocating most of its runs to strong configurations: it starts with a single run per configuration and incrementally performs more runs for promising configurations. This means that it can often afford a large number of runs for the best configurations while rejecting most poor configurations based on a few runs. There is also a guarantee that configurations that were 'unlucky' can be revisited in the search, allowing for a proof that FocusedILS—if run indefinitely—will eventually identify the configuration with the best performance on the entire training set.

Finally, *ParamILS* also implements a mechanism for adaptively choosing the time after which to terminate unsuccessful target algorithm runs. Intuitively, when comparing the performance of two configurations $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ on an instance, and we already know that $\boldsymbol{\theta}_1$ solves the instance in time t_1, we do not need to run $\boldsymbol{\theta}_2$ for longer than t_1: we do not need to know precisely *how bad* $\boldsymbol{\theta}_2$ is, as long as we know that $\boldsymbol{\theta}_1$ is better. More precisely, each comparison of configurations in *ParamILS* is with respect to an instance set $\Pi_{sub} \subset \Pi$, and evaluations of $\boldsymbol{\theta}_2$ can be terminated prematurely when $\boldsymbol{\theta}_2$'s aggregated performance on Π_{sub} is provably worse than that of $\boldsymbol{\theta}_1$. In practice, this so-called *adaptive capping* mechanism can speed up *ParamILS*'s progress by orders of magnitude when the best configuration solves instances much faster than the overall maximal cutoff time [47].

For all experiments in this paper, we used the FocusedILS variant of the most recent publicly available *ParamILS* release 2.3.7[12] with default parameters.

B.2. GGA: Gender-based Genetic Algorithm

The *Gender-based Genetic Algorithm (GGA)* [1] is a configuration procedure that maintains a population of configurations and proceeds according to an evolutionary metaphor, evolving the population over a number of *generations* in which pairs of

[11] The code for computing these features is available at http://www.cs.ubc.ca/labs/beta/Projects/EPMs/.
[12] http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/.

configurations mate and produce offspring. *GGA* also uses the concept of *gender*: each configuration is labeled with a gender chosen uniformly at random, and when configurations are selected to mate there are separate selection pressures for each gender: configurations from the first gender are selected based on their empirical performance, whereas configurations from the other gender are selected uniformly at random. The second gender thus serves as a pool of diversity, countering premature convergence to a poor parameter configuration.

Unlike *ParamILS*' local search mechanism, *GGA*'s recombination operator for combining the parameter values of two parent configurations can operate directly on numerical parameter domains, avoiding the need for discretization.

Like *ParamILS*, *GGA* implements an adaptive capping mechanism, elegantly combining it with a parallelization mechanism that lets it effectively use multiple processing units. *GGA* only ever evaluates configurations in the selection step for the first gender, and its strategy is to evaluate several candidates in parallel until the first one succeeds. Here, the number of configurations to be evaluated in parallel is taken to be identical to the number of processing units available, $\#units$.[13]

Like the FocusedILS variant of *ParamILS*, *GGA* also implements an "intensification" mechanism for increasing the number of runs N it performs for each configuration over time. Specifically, it keeps N constant in each generation, starting with small N_{start} in the first generation, and linearly increasing N up to a larger N_{target} in generation G_{target} and thereafter; N_{start}, N_{target}, and G_{target}, are parameters of *GGA*.

For all experiments in the CSSC, we used the most recent publicly available version of *GGA*, version 1.3.2.[14] *GGA*'s author Kevin Tierney kindly provided a script to convert the parameter configuration space description for each solver from the competition's pcs format[15] to *GGA*'s native xml format. This script allowed us to run *GGA* for all solvers except those with complex conditionals.

Next to the parameters $\#units$, N_{start}, N_{target}, and G_{target} mentioned above, free *GGA* parameters include the maximal number of generations, G_{max} and the size of the population, P_{size}. The setting of these parameters considerably affects *GGA*'s behavior and also determines its overall runtime (when run to completion). If there is an external fixed time budget (as in the CSSC), these parameters can be modified to ensure that GGA does not finish far too early (thus not making effective use of the available configuration budget) while simultaneously ensuring that runs do not take far too long (in which case configuration would be cut off in one of the first generations, where the search is basically still random sampling). It is thus important to set GGA's parameters carefully. We set the following parameters to values hand-chosen by Kevin Tierney for the CSSC (leaving all other parameters at their default values): $\#units = 4$, $P_{size} = 50$, $G_{target} = 75$, $G_{max} = 100$, $N_{start} = 4$, $N_{target} = \#$(training instances in the scenario).[16]

We performed a post hoc analysis, which suggests that these parameters may yet not be optimal: *GGA* often finished relatively few generations within its configuration budget. It might thus make sense to use a smaller value of N_{target} in the future to reduce the number of instances considered per configuration. However, this means that *GGA* would never consider all instances and may overtune as a result. How to best set *GGA*'s parameters is therefore an open research question.

B.3. SMAC: sequential model-based algorithm configuration

In contrast to the model-free configurators *ParamILS* and *GGA*, *SMAC* [46] is a sequential model-based algorithm configuration method, which means that it uses predictive models of algorithm performance [49] to guide its search for good configurations. More specifically, it uses previously observed ⟨configuration, performance⟩ pairs $\langle \boldsymbol{\theta}, f(\boldsymbol{\theta}) \rangle$ to learn a random forest of regression trees (see, *e.g.*, [22]) that express a function $\hat{f} : \boldsymbol{\Theta} \to \mathbb{R}$ predicting the performance of arbitrary parameter configurations (including those not yet evaluated) and then uses this function to guide its search. When instance characteristics $x_\pi \in \mathcal{F}$ are available for each problem instance π, *SMAC* uses observed ⟨configuration, instance characteristic, performance⟩ triplets $\langle \boldsymbol{\theta}, x_\pi, f(\boldsymbol{\theta}, \pi) \rangle$ to learn a function $\hat{g} : \boldsymbol{\Theta} \times \mathcal{F} \to \mathbb{R}$ that predicts the performance of arbitrary parameter configurations on instances with arbitrary characteristics. These so-called *empirical performance models* [49] are then marginalized over the instance characteristics of all training benchmark instances in order to derive the function $\hat{f}$ that predicts average performance for each parameter configuration: $\hat{f}(\boldsymbol{\theta}) = \mathbb{E}_{\pi \sim \Pi_{train}}\left[\hat{g}(\boldsymbol{\theta}, \pi)\right]$.

This performance model is used in a sequential optimization process as follows. After an initialization phase, *SMAC* iterates the following three steps: (1) use the performance measurements observed so far to fit a marginal random forest model $\hat{f}$; (2) use $\hat{f}$ to select a promising configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ to evaluate next, trading off exploration in new parts of the configuration space and exploitation in parts of the space known to perform well; and (3) run $\boldsymbol{\theta}$ on one or more benchmark instances and compare its performance to the best configuration observed so far.

SMAC employs a similar criterion as FocusedILS to determine how many runs to perform for each configuration, and for finite configuration spaces in the limit it also provably converges to the best configuration on the training set. Unlike *ParamILS*, *SMAC* does not require that the parameter space be discretized.

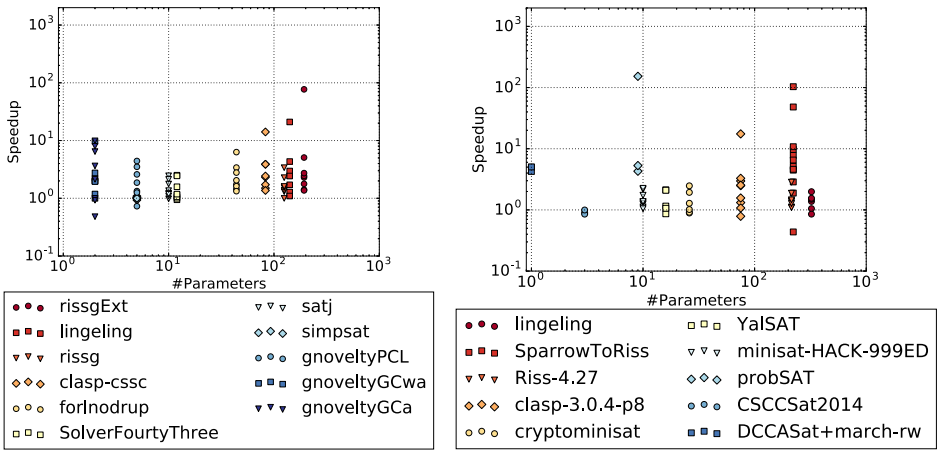
---

[13] This coupling of adaptive capping and parallelization is the reason that *GGA* should not be run on a single core if the objective is to minimize runtime.

[14] https://wiwi.uni-paderborn.de/dep3/entscheidungsunterstuetzungssysteme-und-operations-research-jun-prof-dr-tierney/research/source-code/.

[15] http://aclib.net/cssc2014/pcs-format.pdf.

[16] Actually, due to a miscommunication, we first ran experiments with $N_{target} = 2000$, obtaining somewhat worse results than reported here. After double-checking with Kevin Tierney we then re-ran everything with the correct value of N_{target} that depended on the number of training instances in each configuration scenario. We only report these latter results here.



(a) CSSC 2013 (b) CSSC 2014

Fig. D.11. Same as Fig. 9, but using PAR-1 instead of PAR-10 score.

When used to optimize target algorithm runtime, *SMAC* implements an adaptive capping mechanism similar to the one used in *ParamILS*. When this capping mechanism prematurely terminates an algorithm run we only observe a lower bound of the algorithm's runtime. In order to construct predictive models of algorithm runtime in the presence of such so-called *right-censored data points*, *SMAC* applies model-building techniques derived from the survival analysis literature [45].

Appendix C. Hors-concours solver *Riss3gExt*

So far, we have limited our analysis to the ten open-source solvers that competed for medals. Recall that one additional solver, *Riss3gExt*, only participated *hors concours*. It was not eligible for a medal, because it had been submitted as closed source, being based on a highly experimental code branch of *Riss3g* that had not been exhaustively tested and was therefore likely to contain bugs.

As discussed in Section 2.1, our experimental protocol included various safeguards against such bugs: we measured runtime and memory externally, compared reported solubility status against true solubility status where this was known, and checked returned models when an instance was reported satisfiable. Our configuration pipeline detected and penalized these crashes automatically, enabling the configuration procedures to continue their search and find *Riss3gExt* configurations with no or few crashes. In fact, the final best configurations identified by our configuration pipeline performed very well and would have handily won both the industrial and the crafted track of the CSSC 2013 had *Riss3gExt* been submitted as open source: in the industrial track, it only left 82 problem instances unsolved (compared to 115 for *Lingeling*); and in the crafted track only 44 (compared to 96 for *Clasp-3.0.4-p8*). Even though most of the instances *Riss3gExt* did not solve were due to it crashing, all of these were 'legal' crashes that simply did not output a solution (such as segmentation faults). In particular, we never observed *Riss3gExt* to produce an incorrect output for a CSSC test instance with known satisfiability status.

However, empirical tests with benchmark instances are of course no substitute for formal correctness guarantees, and even seasoned solvers can have bugs. Indeed, after the competition, *Riss3gExt*'s developer found a bug in it (in on-the-fly clause improvement [38]) that caused some satisfiable instances to be incorrectly labeled as unsatisfiable.[17] This being the case, it was fortunate that *Riss3gExt* was ineligible for medals.

While empirical testing on benchmark instances, as done in a competition, can never guarantee the correctness of a solver, in future CSSCs, we consider tightening solubility checks on the benchmark instances used, by either limiting the benchmark sets to contain only instances with known satisfiability status or to require (and check) proofs of unsatisfiability, as in the certified UNSAT track of the SAT competition.

Appendix D. Additional results with PAR-1 score

Fig. D.11 visualizes runtime speedups obtained for each solver, counting timeouts at the cutoff time as the cutoff time itself (PAR-1). Compared to the PAR-10 results in Fig. 9, speedups with PAR-1 are up to a factor of ten smaller for benchmark/solver combinations with many timeouts for the default, but otherwise results are qualitatively similar.

[17] Personal communication with *Riss3gExt*'s developer Norbert Manthey.

References

[1] C. Ansótegui, M. Sellmann, K. Tierney, A gender-based genetic algorithm for the automatic configuration of algorithms, in: I. Gent (Ed.), Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming, CP'09, in: Lecture Notes in Computer Science, vol. 5732, Springer-Verlag, 2009, pp. 142–157.
[2] G. Audemard, L. Simon, Predicting learnt clauses quality in modern SAT solvers, in: [21], 2009, pp. 399–404.
[3] D. Babić, A. Hu, Structural abstraction of software verification conditions, in: W. Damm, H. Hermanns (Eds.), Proceedings of the International Conference on Computer Aided Verification, CAV'07, in: Lecture Notes in Computer Science, vol. 4590, Springer, 2007, pp. 366–378.
[4] D. Babić, A.J. Hu, Exploiting shared structure in software verification conditions, in: K. Yorav (Ed.), Proceedings of the International Conference on Hardware and Software: Verification and Testing, HVC'08, in: Lecture Notes in Computer Science, vol. 4899, Springer, 2008, pp. 169–184.
[5] D. Babić, F. Hutter, Spear theorem prover. Solver description, SAT competition, 2007.
[6] V. Balabanov, Solver43, in: [7], 2013, p. 86.
[7] A. Balint, A. Belov, M. Heule, M. Järvisalo (Eds.), Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2013-1, University of Helsinki, 2013.
[8] A. Balint, N. Manthey, SparrowToRiss, in: [13], 2014, pp. 77–78.
[9] A. Balint, U. Schöning, Choosing probability distributions for stochastic local search and the role of make versus break, in: [25], 2012, pp. 16–29.
[10] R.J. Bayardo Jr., R. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: B. Kuipers, B. Webber (Eds.), Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI'97, AAAI Press, 1997, pp. 203–208.
[11] S. Bayless, D. Tompkins, H. Hoos, Evaluating instance generators by configuration, in: P. Pardalos, M. Resende (Eds.), Proceedings of the Eighth International Conference on Learning and Intelligent Optimization, LION'14, in: Lecture Notes in Computer Science, Springer-Verlag, 2014.
[12] J. Bebel, H. Yuen, Hard SAT instances based on factoring, in: [7], 2013, p. 102.
[13] A. Belov, D. Diepold, M. Heule, M. Järvisalo (Eds.), Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions, Department of Computer Science Series of Publications B, vol. B-2014-2, University of Helsinki, 2014.
[14] D.L. Berre, A. Parrain, The Sat4j library, release 2.2, system description, J. Satisf. Boolean Model. Comput. 7 (2010) 59–64.
[15] A. Biere, The AIGER and-inverter graph (AIG) format, Available at fmv.jku.at/aiger, 2007.
[16] A. Biere, Lingeling, plingeling and treengeling entering the SAT competition 2013, in: [7], 2013, pp. 51–52.
[17] A. Biere, Yet another local search solver and Lingeling and friends entering the SAT competition 2014, in: [13], 2014, pp. 39–40.
[18] A. Biere, A. Cimatti, K.L. Claessen, T. Jussila, K. McMillan, F. Somenzi, Benchmarks from the 2008 hardware model checking competition (HWMCC'08), Available at http://fmv.jku.at/hwmcc08/benchmarks.html, 2008.
[19] A. Biere, A. Cimatti, E. Clarke, M. Fujita, Y. Zhu, Symbolic model checking using SAT procedures instead of BDDs, in: Proceedings of Design Automation Conference, DAC'99, 1999, pp. 317–320.
[20] A. Biere, D. Le Berre, E. Lonca, N. Manthey, Detecting cardinality constraints in CNF, in: C. Sinz, U. Egly (Eds.), Proceedings of the Seventeenth International Conference on Theory and Applications of Satisfiability Testing, SAT'14, in: Lecture Notes in Computer Science, vol. 8561, Springer-Verlag, 2014, pp. 285–301.
[21] C. Boutilier (Ed.), Proceedings of the 22th International Joint Conference on Artificial Intelligence, IJCAI'09, 2009.
[22] L. Breimann, Random forests, J. Mach. Learn. Res. 45 (2001) 5–32.
[23] R. Brummayer, F. Lonsing, A. Biere, Automated testing and debugging of SAT and QBF solvers, in: [25], 2012, pp. 44–57.
[24] C. Cadar, D. Dunbar, D.R. Engler, Klee: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, vol. 8, 2008, pp. 209–224.
[25] A. Cimatti, R. Sebastiani (Eds.), Proceedings of the Fifteenth International Conference on Theory and Applications of Satisfiability Testing, SAT'12, Lecture Notes in Computer Science, vol. 7317, Springer-Verlag, 2012.
[26] E. Clarke, D. Kroening, F. Lerda, A tool for checking ANSI-C programs, in: Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2004, pp. 168–176.
[27] S. Cook, The complexity of theorem proving procedures, in: M. Harrison, R. Banerji, J. Ullman (Eds.), Proceedings of the Third Annual ACM Symposium on the Theory of Computing, STOC'71, ACM, 1971, pp. 151–158.
[28] J. Crawford, A. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: Proceedings of the National Conference on Artificial Intelligence, AAAI'94, AAAI Press/MIT Press, 1994, pp. 1092–1097.
[29] T.-T. Duong, D.-N. Pham, gNovelty+GC: weight-enhanced diversification on stochastic local search for SAT, in: [7], 2013, pp. 49–50.
[30] N. Eén, N. Sörensson, An extensible SAT-solver, in: E. Giunchiglia, A. Tacchella (Eds.), Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing, SAT'03, in: Lecture Notes in Computer Science, vol. 2919, Springer, 2003, pp. 502–518.
[31] A. Fern, R. Khardon, P. Tadepalli, The first learning track of the international planning competition, Mach. Learn. 84 (1–2) (2011) 81–107.
[32] M. Feurer, A. Klein, K. Eggensperger, J.T. Springenberg, M. Blum, F. Hutter, Efficient and robust automated machine learning, in: C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett (Eds.), Proceedings of the 29th International Conference on Advances in Neural Information Processing Systems, NIPS'15, 2015.
[33] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: from theory to practice, Artif. Intell. 187–188 (2012) 52–89.
[34] A.V. Gelder, Toward leaner binary-clause reasoning in a satisfiability solver, Ann. Math. Artif. Intell. 43 (1) (2005) 239–253.
[35] C. Gomes, B. Selman, N. Crato, H. Kautz, Heavy-tailed phenomena in satisfiability and constraint satisfaction problems, J. Autom. Reason. 24 (1–2) (2000) 67–100.
[36] C.-S. Han, J.-H. Jiang, Simpsat 1.0 for sat challenge 2012, in: A. Balint, A. Belov, D. Diepold, S. Gerber, M. Järvisalo, C. Sinz (Eds.), Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions, in: Department of Computer Science Series of Publications B, vol. B-2012-2, University of Helsinki, 2012, p. 59.
[37] C.-S. Han, J.-H. Jiang, When Boolean satisfiability meets Gaussian elimination in a simplex way, in: Computer Aided Verification, 2012, pp. 410–426.
[38] H. Han, F. Somenzi, On-the-fly clause improvement, in: [56], 2009, pp. 209–222.
[39] M.J. Heule, M. Järvisalo, A. Biere, Efficient CNF simplification based on binary implication graphs, in: [74], 2011, pp. 201–215.
[40] H. Hoos, T. Stützle, Stochastic Local Search: Foundations & Applications, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.
[41] H.H. Hoos, Programming by optimization, Commun. ACM 55 (2) (2012) 70–80.
[42] F. Hutter, D. Babić, H. Hoos, A. Hu, Boosting verification by automatic tuning of decision procedures, in: L. O'Conner (Ed.), Formal Methods in Computer Aided Design, FMCAD'07, IEEE Computer Society Press, 2007, pp. 27–34.
[43] F. Hutter, A. Balint, S. Bayless, H. Hoos, K. Leyton-Brown, Results of the Configurable SAT Solver Challenge 2013, Technical report 276, University of Freiburg, Department of Computer Science, 2014.
[44] F. Hutter, H. Hoos, K. Leyton-Brown, Tradeoffs in the empirical evaluation of competing algorithm designs, in: Special Issue on Learning and Intelligent Optimization, Ann. Math. Artif. Intell. 60 (1) (2010) 65–89.
[45] F. Hutter, H. Hoos, K. Leyton-Brown, Bayesian optimization with censored response data, in: NIPS Workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits, 2011, Published online.

[46] F. Hutter, H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: C. Coello (Ed.), Proceedings of the Fifth International Conference on Learning and Intelligent Optimization, LION'11, in: Lecture Notes in Computer Science, vol. 6683, Springer-Verlag, 2011, pp. 507–523.
[47] F. Hutter, H. Hoos, K. Leyton-Brown, T. Stützle, ParamILS: an automatic algorithm configuration framework, J. Artif. Intell. Res. 36 (2009) 267–306.
[48] F. Hutter, M. Lindauer, S. Bayless, H. Hoos, K. Leyton-Brown, Results of the Configurable SAT Solver Challenge 2014, Technical report 277, University of Freiburg, Department of Computer Science, 2014.
[49] F. Hutter, L. Xu, H. Hoos, K. Leyton-Brown, Algorithm runtime prediction: methods and evaluation, Artif. Intell. 206 (2014) 79–111.
[50] M. Järvisalo, D.L. Berre, O. Roussel, L. Simon, The international SAT solver competitions, AI Mag. 33 (1) (2012).
[51] M. Järvisalo, A. Biere, M.J. Heule, Blocked clause elimination, in: J. Esparza, R. Majumdar (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, in: Lecture Notes in Computer Science, vol. 6015, Springer, Berlin, Heidelberg, 2010, pp. 129–144.
[52] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, M. Sellmann, Algorithm selection and scheduling, in: J. Lee (Ed.), Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming, CP'11, in: Lecture Notes in Computer Science, vol. 6876, Springer-Verlag, 2011, pp. 454–469.
[53] H. Kautz, B. Selman, Pushing the envelope: planning, propositional logic, and stochastic search, in: H. Shrobe, T. Senator (Eds.), Proceedings of the Fourteenth National Conference on Artificial Intelligence, AAAI'96, AAAI Press, 1996, pp. 1194–1201.
[54] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: [13], 2014, pp. 318–325.
[55] A. KhudaBukhsh, L. Xu, H. Hoos, K. Leyton-Brown, SATenstein: automatically building local search SAT solvers from components, in: [21], 2009, pp. 517–524.
[56] O. Kullmann (Ed.), Proceedings of the Twelfth International Conference on Theory and Applications of Satisfiability Testing, SAT'09, Lecture Notes in Computer Science, vol. 5584, Springer, 2009.
[57] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari, The irace Package, Iterated Race for Automatic Algorithm Configuration, Technical report, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.
[58] H. Lourenço, O. Martin, T. Stützle, Iterated local search, in: Handbook of Metaheuristics, Springer, New York, 2003, pp. 321–353.
[59] C. Luo, S. Cai, W. Wu, K. Su, Focused random walk with configuration checking and break minimum for satisfiability, in: C. Bessiere (Ed.), Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, CP'13, in: Lecture Notes in Computer Science, vol. 4741, Springer-Verlag, 2012, pp. 481–496.
[60] C. Luo, S. Cai, W. Wu, K. Su, Double configuration checking in stochastic local search for satisfiability, in: C. Brodley, P. Stone (Eds.), Proceedings of the Twenty-Eighth National Conference on Artificial Intelligence, AAAI'14, AAAI Press, 2014, pp. 2703–2709.
[61] I. Lynce, J.P. Marques-Silva, Probing-based preprocessing techniques for propositional satisfiability, in: 15th IEEE International Conference on Tools with Artificial Intelligence, IEEE Computer Society, 2003, pp. 105–110.
[62] N. Manthey, Coprocessor 2.0 – a flexible CNF simplifier, in: [25], 2012, pp. 436–441.
[63] N. Manthey, The SAT solver RISS3G at SC 2013, in: [7], 2013, pp. 72–73.
[64] N. Manthey, Riss 4.27, in: [13], 2014, pp. 65–67.
[65] N. Manthey, M.J. Heule, A. Biere, Automated reencoding of Boolean formulas, in: A. Biere, A. Nahir, T. Vos (Eds.), Hardware and Software: Verification and Testing, in: Lecture Notes in Computer Science, vol. 7857, Springer, Berlin, Heidelberg, 2013, pp. 102–117.
[66] N. Manthey, T. Philipp, Formula simplifications as DRAT derivations, in: C. Lutz, M. Tielscher (Eds.), KI 2014: Advances in Artificial Intelligence, in: Lecture Notes in Computer Science, vol. 8736, Springer, Berlin, Heidelberg, 2014, pp. 111–122.
[67] N. Manthey, P. Steinke, Too many rooks, in: [13], 2014, pp. 97–98.
[68] F. Mugrauer, A. Balint, SAT encoded graph isomorphism benchmark description, in: [7], 2013.
[69] F. Mugrauer, A. Balint, SAT encoded low autocorrelation binary sequence (labs) benchmark description, in: [7], 2013.
[70] E. Nudelman, K. Leyton-Brown, H.H. Hoos, A. Devkar, Y. Shoham, Understanding random SAT: beyond the clauses-to-variables ratio, in: M. Wallace (Ed.), Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming, CP'04, in: Lecture Notes in Computer Science, vol. 3258, Springer-Verlag, 2004, pp. 438–452.
[71] C. Oh, Minisat hack 999ed, minisat hack 1430ed and swdia5by, in: [13], 2014, p. 46.
[72] M. Prasad, A. Biere, A. Gupta, A survey of recent advances in SAT-based formal verification, Int. J. Softw. Tools Technol. Transf. 7 (2) (2005) 156–173.
[73] O. Roussel, Controlling a solver execution with the runsolver tool, J. Satisf. Boolean Model. Comput. 7 (4) (2011) 139–144.
[74] K.A. Sakallah, L. Simon (Eds.), Proceedings of the Fourteenth International Conference on Theory and Applications of Satisfiability Testing, SAT'11, Lecture Notes in Computer Science, vol. 6695, Springer, 2011.
[75] J. Seipp, S. Sievers, F. Hutter, Fast downward SMAC. Planner abstract, IPC 2014 Planning and Learning Track, 2014.
[76] L. Simon, D.L. Berre, E. Hirsch, The SAT2002 competition report, Ann. Math. Artif. Intell. 43 (2005) 307–342.
[77] M. Soos, CryptoMiniSat v4, in: [13], 2014, p. 23.
[78] M. Soos, K. Nohl, C. Castelluccia, Extending SAT solvers to cryptographic problems, in: [56], 2009, pp. 244–257.
[79] P. Stephan, R. Brayton, A. Sangiovanni-Vencentelli, Combinational test generation using satisfiability, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 15 (1996) 1167–1176.
[80] C. Thornton, F. Hutter, H. Hoos, K. Leyton-Brown, Auto-WEKA: combined selection and hyperparameter optimization of classification algorithms, in: I. Dhillon, Y. Koren, R. Ghani, T. Senator, P. Bradley, R. Parekh, J. He, R. Grossman, R. Uthurusamy (Eds.), The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'13, ACM Press, 2013, pp. 847–855.
[81] D. Tompkins, A. Balint, H. Hoos, Captain jack: new variable selection heuristics in local search for SAT, in: [74], 2011, pp. 302–316.
[82] J. Torán, On the resolution complexity of graph non-isomorphism, in: M. Järvisalo, A. Van Gelder (Eds.), Proceedings of the Sixteenth International Conference on Theory and Applications of Satisfiability Testing, SAT'13, in: Lecture Notes in Computer Science, vol. 7962, Springer-Verlag, 2013, pp. 52–66.
[83] A. van Gelder, Another look at graph coloring via propositional satisfiability, in: Proceedings of Computational Symposium on Graph Coloring and Generalizations, COLOR-02, 2002, pp. 48–54.
[84] L. Xu, F. Hutter, H. Hoos, K. Leyton-Brown, SATzilla: portfolio-based algorithm selection for SAT, J. Artif. Intell. Res. 32 (2008) 565–606.
[85] E. Zarpas, Benchmarking SAT solvers for bounded model checking, in: F. Bacchus, T. Walsh (Eds.), Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing, SAT'05, in: Lecture Notes in Computer Science, vol. 3569, Springer, 2005, pp. 340–354.