EPiC Series in Computing

Volume 59, 2019, Pages 111–126

Proceedings of Pragmatics of SAT 2015 and 2018

The Effect of Scrambling CNFs*

Armin Biere[1] and Marijn J.H. Heule[2]

[1] Johannes Kepler University Linz
[2] University of Texas at Austin

Abstract

It has been an ongoing, decades-long debate about how SAT solvers and in general different or new algorithms should be evaluated and compared both in competitions and more importantly in papers. Evaluations are usually performed on existing benchmarks. Cross-validation and other means to avoid over-fitting are rarely used. In this paper we revisit the old idea of scrambling benchmarks also used in early competitions. Scrambling has the goal to make results of such evaluations more robust. We present a new method for scrambling CNFs, which allows to gradually increase the effect of scrambling, from keeping the scrambled CNF close to the original CNF, to complete random permutation of variables, clauses, and phases of literals. We used this method to scramble benchmarks from the last two SAT competitions and solved them with the best solvers in the main track of the last SAT competition. As expected our experimental results suggest that scrambling has a substantial effect on the performance of individual solvers but surprisingly has little effect on rankings among solvers. As a consequence we argue that only using our method of scrambling is not enough to increase robustness of competitions and evaluations in general.

Introduction

In the SAT 2003 Competition complete random permutation of variables and clauses was used, and random flipping of literals too. This randomization of instances was also called "shuffling". In the analysis of the competition it was observed [6] that such scrambling might result in totally different run-times of the same solver on the original and the scrambled instance (called the *lisa* syndrome in [6]). Triggered by this observation the SAT 2004 Competition report [7] includes experiments of running solvers in the "industrial" track on original as well as on two scrambled versions for each instance. Solvers performed worse on scrambled benchmarks.

A possible explanation might be that solvers simply use the given variable order for picking the first decisions, usually implicitly by the way how the binary heap acting as decision queue is initialized. Since this order often encodes some structural property of the original problem, one often gets "good" initial decisions this way, which in the end improves performance.

It was also claimed that better performing solvers are less robust with respect to scrambling. Thus scrambling became questionable to measure performance of SAT solvers on industrial benchmarks and as a consequence was abandoned. However, SAT solvers have become

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE).

substantially more robust since 2003. It is therefore expected that the impact of scrambling on the performance on current state-of-the-art solvers is less profound.

One crucial change since 2005 has been the use of bounded variable elimination [10]. Actually, eliminating variables somewhat scrambles the formula, but this technique has been highly effective and easily overcomes its negative effects. Another robustness enhancement has been the use of phase-saving combined with rapid restarts [17] (2007). Early CDCL solvers used negative branching, i.e., assigning decision variables always to false [11]. This was more or less an heuristic choice based on how problems are generally encoded. Negative branching is much more effective compared to assigning a random truth value to decision variables. Scrambling instances would turn the former into the latter, while it has almost no influence on phase-saving.

Another motivation is that scrambling demonstrates the upper bound on how much a solver is expected to gain form adding noise. Quite some papers present a technique and claim that it improves performance based one solving a handful of instances more after adding the new technique. This is argued to be progress as the difference in solved formulas between the top solvers during the competition is also only a handful of instances. We will show that this reasoning is problematic: for all top solvers it is possible to increase the number of solved instances by simply scrambling them with the same strategy and the same seed. As a consequence, papers should be able to show that adding a new technique outperforms various scrambling strategies. Our scrambling tool can be used to strengthen the claim that a certain technique is effective.

Scrambling has also been proposed as a method to avoid cheating by just hashing instances to their status. We are not aware of any attempt in this direction. The requirement to provide witnesses and now even proofs makes cheating by hashing extremely difficult, especially for unsatisfiable formulas. Yet to some degree such hashing might happen unintentionally: Solver parameters are optimized to solve as many existing benchmarks as possible, which in turn may result in storing good decisions for known formulas. Scrambling could counter this effect.

Understanding the effects of completely random "shuffling" was the motivation for [15] which reports similar results as [7] but also attempts a more precise statistical analysis. Already in [1] it was also observed that "shuffling" (so complete random scrambling) was less harmful to solver performance than expected and occasionally even lead to more solved instances. We are going to apply scrambling to the current state-of-the-art, purely from an experimental perspective, apply current much longer time outs and also propose and experiment with variants of scrambling which have a tunable randomization effect.

Scrambling

First of all, it is important to note, that we are working with the DIMACS format, where literals are represented as integers. Consider the following CNFs over two variables in DIMACS format which all have exactly one solution (first three "`v -1 -2 0`" and last one "`v -1 2 0`"):

```
                c only          c only          c clauses reversed
                c variables     c clauses       c first literal flipped
c original      c swapped       c reversed      c variables swapped
p cnf 2 3       p cnf 2 3       p cnf 2 3       p cnf 2 3
-1 -2 0         -2 -1 0         1 -2 0          -2 -1 0
-1 2 0          -2 1 0          -1 2 0          2 1 0
1 -2 0          2 -1 0          -1 -2 0         2 -1 0
```

These examples introduce all the forms of scrambling we consider in this paper:

(i) permuting variables, (ii) permuting clauses, and (iii) flipping literals.

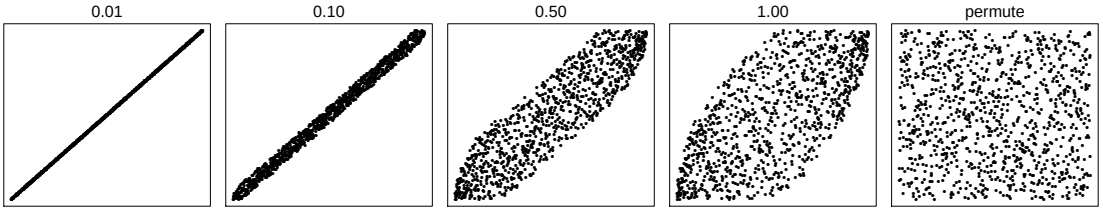

Figure 1: From 1% over 10% and 50% to 100% move window and complete random permutation. The original position i is on the x-axis and new position $\pi(i)$ on the y-axis.

Note that we do not permute literal positions within clauses at this point. There are also other more sophisticated forms of scrambling, like removing or adding clauses or even variables and in general of course any kind of transformations, even though probably only satisfiability preserving transformations are interesting. The reason for restricting our discussion to permuting and flipping is because this model has already been used in earlier competitions and that even for this simple form of scrambling the empirical effect is considered unclear, particularly with respect to how scrambling would effect competition results for state-of-the-art solvers.

Variables and clauses were always permuted in a complete random way in earlier work on "shuffling" CNFs. Similarly literals were flipped with 50% probability. It was also argued that this random process would destroy the structure of the CNF and thus make the formula (unnecessarily) harder. Thus as a starting point of the work reported in this paper we were searching for a tunable way of randomizing CNFs, with the property that the amount of randomization can be specified explicitly. The hope was that such "light" randomization would avoid over-tuning effects without destroying the structure too much, i.e., keeping solving time of scrambled instances close to solving time of the original formula.

Our scrambling process[1] works the same for both variables and clauses. Each variable and clause is just abstracted away to the position where it occurs in the input. Assume we want to scramble either n variables or n clauses and we are also given $w \in \mathbb{R}$ with $0 \leq w$, called the *relative move window size*. Let $S = 0, \ldots, n-1$ be the sequence of the first n natural numbers. We need another sequence of (in principle) real numbers $d_0, \ldots, d_{n-1} \in [0, 1)$, i.e., $0 \leq d_i < 1$, for $i \in S$. These are for instance generated by a random number generator such as `drand48`, which further can be assumed to be parameterized by a initial random seed (through `srand48`). The result will be a permutation of $\pi : S \to S$ which for all $0 \leq i, j < n$ satisfies

$$\text{if} \quad i + w \cdot n \cdot d_i < j + w \cdot n \cdot d_j \quad \text{then} \quad \pi(i) < \pi(j)$$

To compute this permutation we sort S with respect to $i + w \cdot n \cdot d_i$. Applied to n variables the relative move window w determines the number $w \cdot n$ of variables a variable at position i can "overtake" in the new order $(\pi(0), \pi(1), \ldots)$. The effect of varying w is shown in Fig. 1. We obtain a completely random permutation if we only require $\pi(i) < \pi(j)$ whenever $d_i < d_j$.

Pseudo-code for this scrambling procedure can be found in Fig. 2. It is applied to the number n of variables as first parameter and returns a table representing a permutation of the variable indices $0, \ldots, n-1$. Note that the DIMACS format requires to shift the start of the sequence by one. The same idea is used to permute the input clauses.

If the third parameter "`bool permute`" is true then the target position "`pos`" is set to a random number d with $0 \leq d < 1$ and thus sorting w.r.t. this target positions gives a com-

[1] A reviewer suggested a simpler way to achieve the same effect, but we were not able to investigate this proposal yet and particularly were not able to re-run the experiments for the workshop version of this paper.

```
int[] scramble (int n, double w, bool permute)
    struct { int src; double pos; } order[n];
    for (int i = 0; i < n; i++)  order[i].src = i;
    if (permute)
        for (int i = 0; i < n; i++)
            order[i].pos = drand48 ();
    else
        for (int i = 0; i < n; i++)
            order[i].pos = i + w * n * drand48 ();
    sort (order  w.r.t.  pos);
    int res[n];
    for (int i = 0; i < n; i++)  res[i] = order[i].src;
    return res;
```

Figure 2: Pseudo code for tunable scrambling of variables and clauses.

abbreviation	**solver name in competition**		**last names of authors**
maplelcm	`Maple_LCM_Dist`	[18]	Xiao, Luo, Li, Manyà, Lü
maplecompsps	`MapleCOMSPS_LRB_VSIDS_2_drup`	[13]	Liang, Oh, Ganesh, Czarnecki, Poupart
comspspulsar	`COMiniSatPS_Pulsar_drup`	[16]	Oh
cadical	`cadical-sc17-proof`	[8]	Biere
tchglucose	`tch_glucose3`	[14]	Moon, Mary
glucose41	`glucose-4.1`	[2]	Audemard, Simon
gluvc	`glu_vc`	[9]	Chen

Table 1: Mapping of abbreviated names to the ones used in the competition.

plete random permutation (called "shuffling" in earlier work). Otherwise the second parameter "`double w`" determines the amount of scrambling. If it is zero then the result is the identity function. As larger it gets as closer the result is to a completely randomly picked permutation. If it is for instance 0.01 then variables are respositioned randomly within a window of 1% of all variables. The plots in Fig. 1 show this effect in the first four plots where `permute` is false. The last plot shows a random permutation where `permute` is true. They were generated from applying our scrambler `scranfilize` to a concrete CNF with 1035 variables and dumping the variable permutation table.

Experiments

In our experiments we focus on the top 17 best performing solvers in the main track of the SAT 2017 Competition [5] out of 29 ranked configurations. We then selected for each group of submitters the best performing solver (configuration) and ended up with 7 solvers listed in Table 1. This selection is the same for both type of scores considered ("par2" and "solved").

our experiments					SAT 2017 Competition					
rank	par2	solved	sat	uns	rank	par2	solved	sat	uns	
1	1639735	208	101	107	1	1610934	208	102	106	maplelcm
2	1823497	184	90	94	2	1780711	188	93	95	maplecompsps
3	1868096	181	87	94	3	1798300	188	89	99	comspspulsar
4	1877512	180	83	97	4	1825427	185	83	102	cadical
5	1961085	172	77	95	5	1890486	179	80	99	tchglucose
6	1969231	171	79	92	6	1893632	180	85	95	glucose41
7	2037362	165	73	92	7	1958463	174	77	97	gluvc

Table 2: Comparison of our original experiments with competition results.

Table 1 also contains the mapping of the abbreviated names used in this paper to those longer and more cumbersome names used in the competition. The solver names are (for these solvers) the same as the name of the zip file of the source code available from the SAT 2017 Competition webpage which also matches the directory names if unzipped.

The solvers are ranked by performance. As it turns out, both considered scores "par2" and "solved" give the same ranking in our experiments. In earlier SAT competitions solvers were ranked by the "solved" score, which is the number of solved instances within in a time limit T (in recent competition set to 5000 seconds). The "par2" score is well known from the literature too and has been used in the SAT 2017 Competition to put more emphasis on speed. It is computed as the sum of the running times a solver took to solve all instances in the given benchmark set (for instance 350 benchmarks from the main track of the SAT 2017 Competition). However, each unsolved instance contributes two times the time limit ($2 \cdot T$), thus essentially assuming that in these unsolved cases the solver would have solved the instance in (exactly) twice the time limit.

In our experiments we used the same time limit of 5000 seconds but slightly different compute nodes than in the SAT 2017 Competition. Each dual socket compute node in our cluster had two Intel Xeon E5-2620 v4 CPUs running at 2.10GHz with turbo-mode disabled, while the STAREXEC nodes used in the competition had Intel Xeon E5-2609 CPUs running at 2.40GHz. We were running 16 jobs (solver benchmark pairs) in parallel on each node limiting main memory per job to 7 GB (using the "runlim" tool). Note, that our compute nodes have 128 GB shared main memory each and $16 \cdot 7$ GB $= 112 < 128$ GB. In our experience this set-up only leads to minor performance differences between repeated experiments. The SAT 2017 Competition used a much higher memory limit 24 GB though. However, in our experiments with the SAT 2017 Competition benchmarks only the best performing solver "maplelcm" reached that limit and also for at most two benchmarks ("T106.2.0" and "T50.2.0"). These cases were of course treated as unsolved instances.

Table 2 compares our results of running the selected solvers on the original unscrambled benchmarks on our hardware versus the published[2] competition results. Our cluster is slightly slower, but rankings are the same for the selected solvers.

We used the same source code and build scripts as submitted to the competition. As required to enter the main track the default for all these solvers is to produce DRAT [12] proofs, which is also not easy to disable. Thus we kept proof generation enabled but modified provided execution scripts to write proofs to "`/dev/null`", which accordingly were also not checked.

The different forms of scrambling used in the experiments are explained in Tab. 3. Those

[2] https://baldur.iti.kit.edu/sat-competition-2017/index.php?cat=results

		permute variables -p	permute clauses -P	reverse variables -r	reverse clauses -R	flip probability literals -f	relative move window variables -v	relative move window clauses -c
*	orig							
	vrev			1				
	crev				1			
	pf50	1				0.50		
	qf50		1			0.50		
*	bf50	1	1			0.50		
	f001					0.01		
	f010					0.10		
	f050					0.50		
	f100					1.00		
	v001						0.01	
	v010						0.10	
	v100						1.00	
	c001							0.01
	c010							0.10
	c100							1.00
*	a001					0.01	0.01	0.01
	a010					0.10	0.10	0.10
	a100					0.50	1.00	1.00

Table 3: Different forms of scrambling used in the experiments (empty entries 0).

marked with an asterisk "∗" correspond to the most interesting variants (original "orig", shuffled "bf50", light "a001"), which we also marked in the plots below by vertical lines. The 4$^{\text{th}}$ row gives the actual command line options used for our tool `scranfilize`. Beside completely randomly permuting we also allow to reverse the order of variables and clauses (not shown in the pseudo-code of Fig. 2). Empty entries in the table correspond to zero and thus disable the corresponding feature. It might be important to note that literals are flipped individually with the given probabiliy. Thus f100 replaces all literals by their negation (flipping all), while f050 flips a literal with 50% probabiliy, which is the most random way of flipping literals (as in pf50, qf50, bf50, a100 and of course f050). The code of our tool `scranfilize` as well as experimental data can be found at http://fmv.jku.at/scranfilize.

Figures 3-6 and related tables 4-5 show the effect of scrambling the benchmarks of the main track of the SAT 2017 Competition on the performance of SAT solvers. The solver "maplelcm" clearly dominates the competition, both on the "par2" score and the number of instances, as well as on satisfiable and unsatisfiable instances—regardless of the scrambling strategy. Thus we used the number of instances solved by "maplelcm" to order the scrambling strategies on the x-axis. This ordering gives a monotonic decrease of the line for "maplelcm" in Fig. 4, but for consistency is also applied to Figures 3-6.

Also, the ranking of solvers is quite robust for all scrambling strategies as shown in Table 4. However, the relative performance on satisfiable vs. unsatisfiable instances can differ substan-

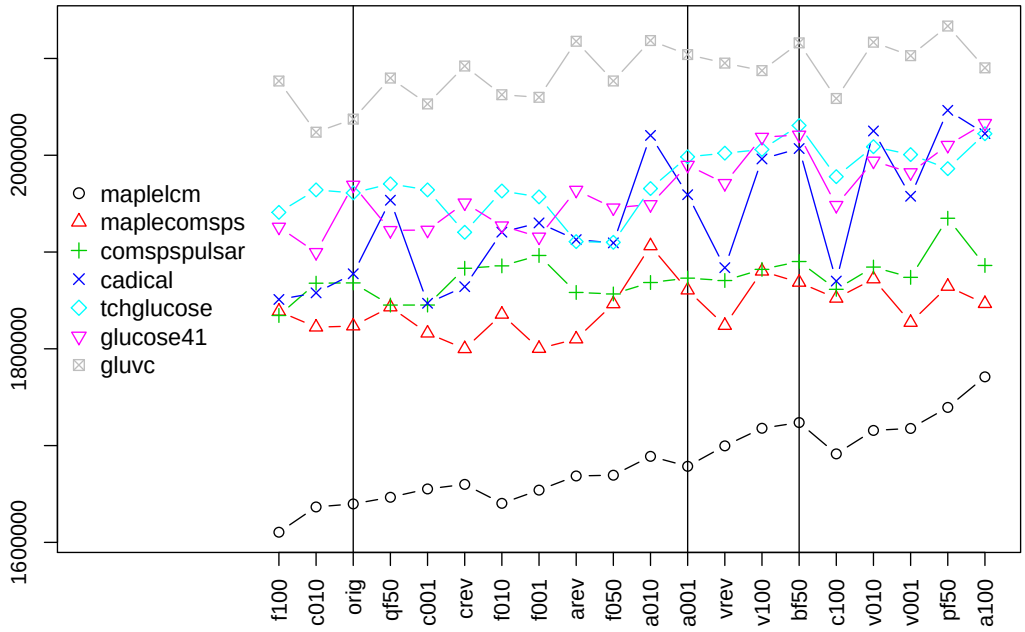

Figure 3: Par2 score (y-axis) for instances from main track of the SAT 2017 Competition after applying different forms of scrambling (x-axis) – smaller is better.

tially. For example, "cadical" ranks on average second on unsatisfiable instances, but fifth on satisfiable ones. See also the cactus plots in Fig. 7-8, which allow to compare the least robust ("cadical") vs. most robust solver ("comspspulsar") for the benchmarks in the main track of the SAT 2017 Competition w.r.t. solved instances, i.e., column "solved σ" in Tab. 5.

Scrambling the instances of the 2017 suite typically reduces solver performance. Yet some strategies actually improve performance. For example, strategy c010 results in equal or better performance of all solvers compared to orig. Notice that "glucose41" even solves 10 additional instances with strategy c010. Many papers that present a new SAT solving technique claim that solving 10 additional instances demonstrates that the technique is useful in practice. Our experiments show that the technique may only cause an effect that is similar to scrambling and thus is not better than adding noise. In order to strengthen a claim that a technique is useful, we propose to include additional evidence that the technique outperforms scrambling. For example, one can run the solver without the technique several times using strategy c010 with different random seeds and show that the solver with the new technique is always stronger.

After analyzing the results on SAT 2017 Competition benchmarks, we were curious whether these findings apply to the SAT 2016 Competition [4, 3] benchmarks too. This is particularly interesting since solvers from 2017 can be considered to have been "trained" on these benchmarks. The remaining figures and tables provide results for SAT 2016 Competition benchmarks using the same solvers. Since results for the main track of the SAT 2016 Competition were separated into "application" and "crafted" instances (in contrast to the SAT 2017 Competition) we have also split figures and tables accordingly but do not include separate plots for "satisfiable" and "unsatisfiable" instances due to space reasons. Instead of the official number of 300 benchmarks(see also [3]), we used only 299 instances in the application track, not considering

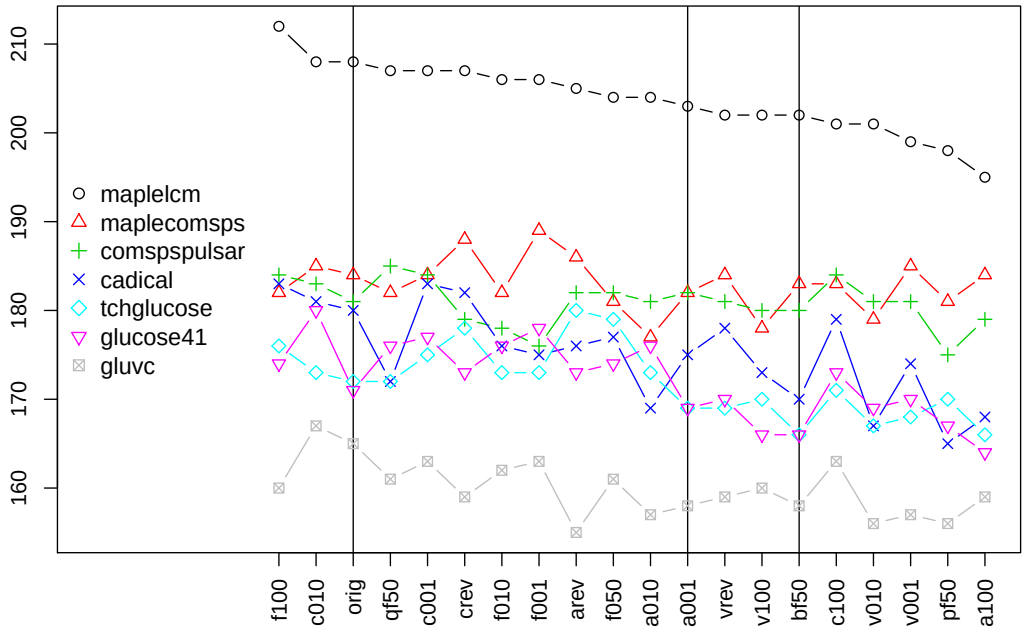

Figure 4: Solved instances (y-axis) from main track of the SAT 2017 Competition after applying different forms of scrambling (x-axis) – larger is better.

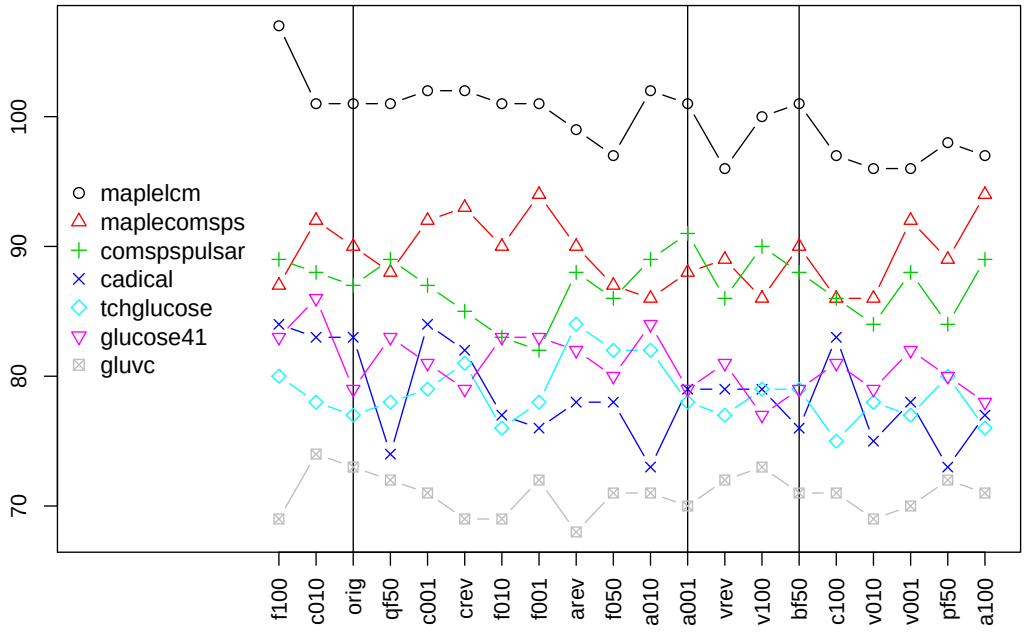

Figure 5: Solved *satisfiable* instances (y-axis) from main track of the SAT 2017 Competition after applying different forms of scrambling (x-axis) – larger is better.

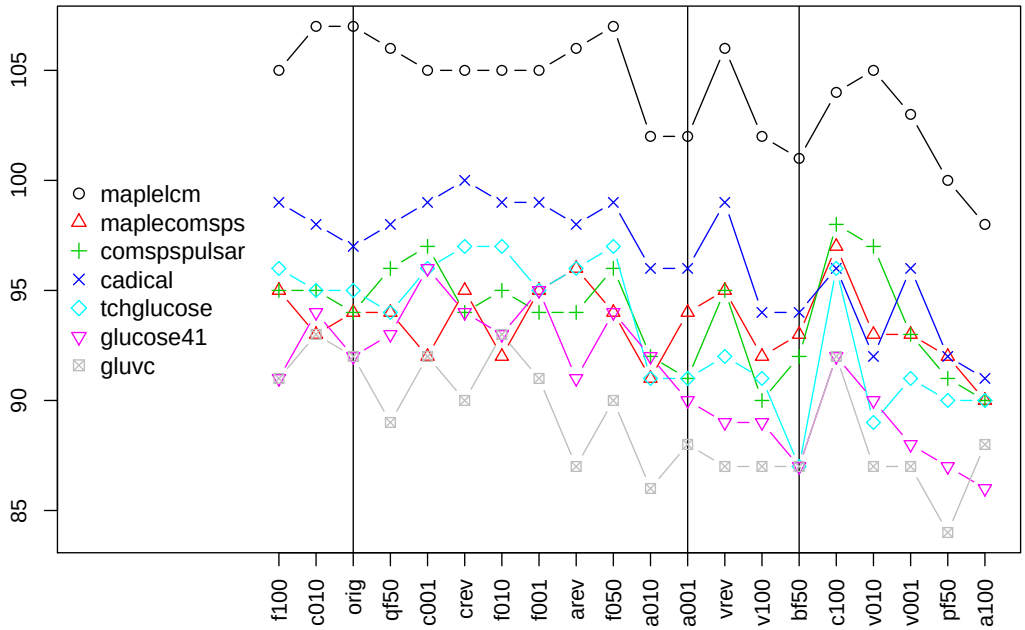

Figure 6: Solved *unsatisfiable* instances (y-axis) from main track of the SAT 2017 Competition after applying different forms of scrambling (x-axis) – larger is better.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
maplelcm	1	1.00	0.00	1	1.00	0.00	1	1.00	0.00	1	1.00	0.00
maplecomsps	2	2.10	0.31	2	2.40	0.60	2	2.30	0.47	4	3.85	1.31
comspspulsar	3	3.00	0.46	3	2.75	0.64	3	2.75	0.55	5	3.95	1.05
cadical	4	4.40	0.88	4	4.40	0.88	4	5.10	0.85	2	2.25	0.64
tchglucose	5	5.40	0.75	5	5.35	0.75	6	5.35	0.75	3	4.50	1.05
glucose41	6	5.10	0.72	6	5.10	0.85	5	4.50	0.76	6	5.55	0.83
gluvc	7	7.00	0.00	7	7.00	0.00	7	7.00	0.00	7	6.90	0.31

Table 4: Original (o) and average (μ) ranking with standard deviation (σ) on benchmarks from the main track of the SAT 2017 Competition.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
maplelcm	1639735	1681268	40656	208	203.85	4.00	101	99.80	2.78	107	104.05	2.50
maplecomsps	1823497	1841931	27838	184	182.95	3.00	90	89.45	2.70	94	93.50	1.73
comspspulsar	1868096	1873304	21828	181	180.90	2.61	87	86.95	2.42	94	93.95	2.35
cadical	1877512	1935606	66136	180	175.15	5.41	83	78.55	3.59	97	96.60	2.72
tchglucose	1961085	1973028	34740	172	172.00	4.09	77	78.70	2.27	95	93.30	3.10
glucose41	1969231	1963012	38947	171	172.10	4.42	79	80.95	2.28	92	91.15	2.85
gluvc	2037362	2085159	29408	165	159.95	3.19	73	70.90	1.59	92	89.05	2.61

Table 5: Original (o) and average (μ) scores with standard deviation (σ) on benchmarks from the main track of the SAT 2017 Competition.

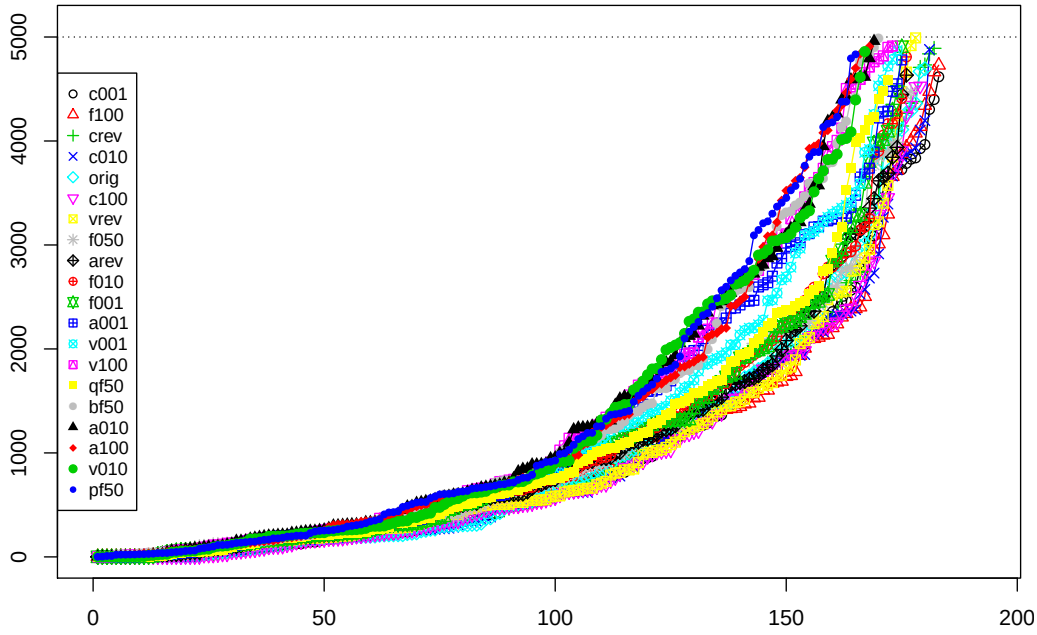

Figure 7: Cactus plot of the main track of the SAT 2017 Competition for the *most sensitive* solver "cadical".

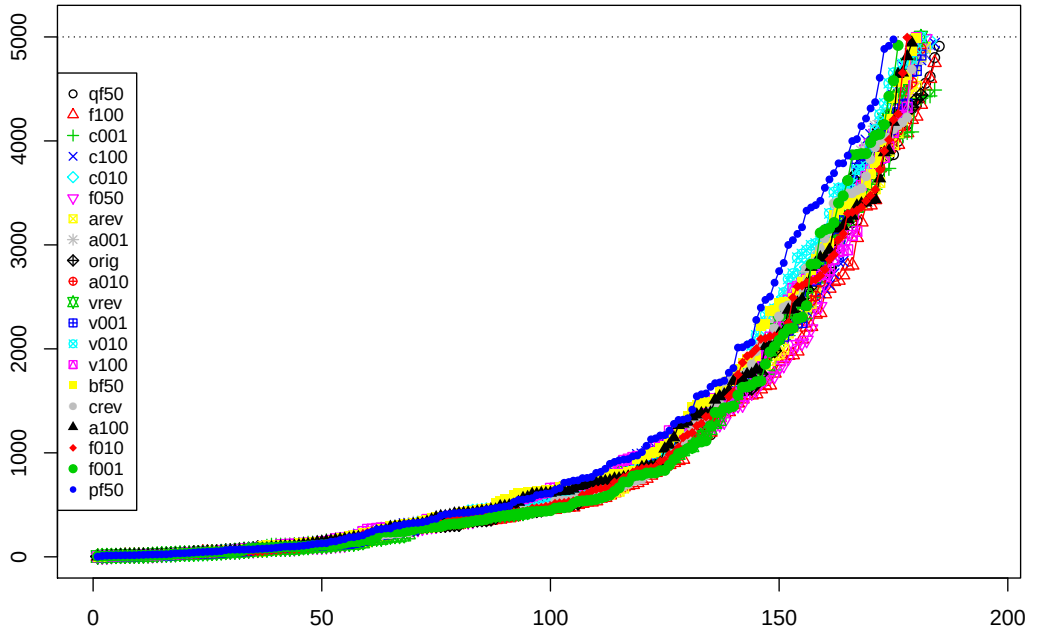

Figure 8: Cactus plot of the main track of the SAT 2017 Competition for the *most robust* solver "comspspulsar".

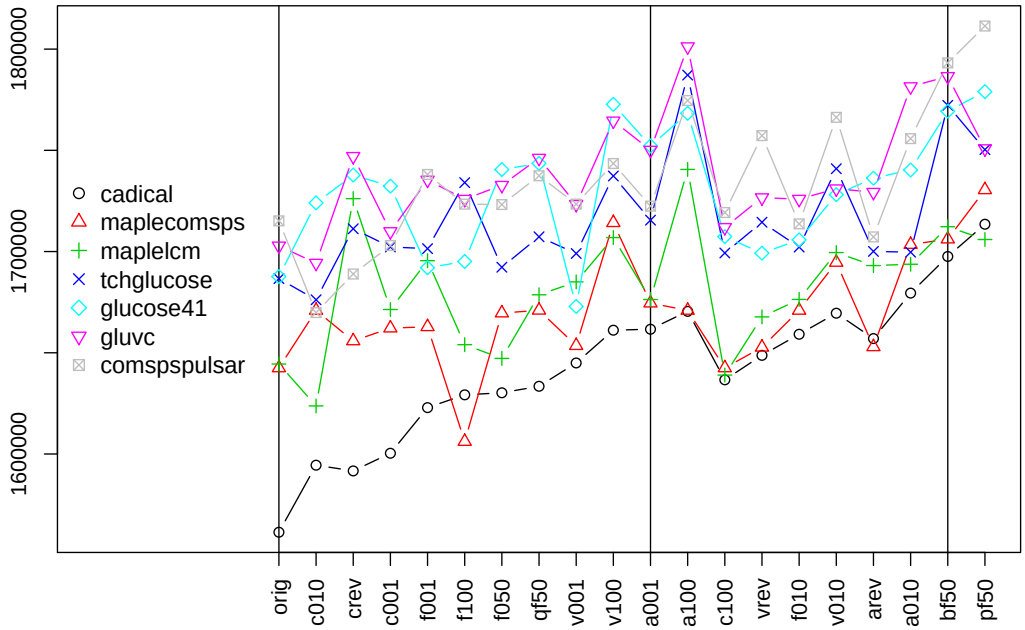

Figure 9: Par2 score (y-axis) for instances from application track of the SAT 2016 Competition after applying different forms of scrambling (x-axis) – smaller is better.

"`sncf_model_ixl_bmc_depth_14`", since first during the competition a truncated syntactically incorrect DIMACS file was used for this benchmark and accordingly all solvers aborted during parsing, and second benchmarks from this family anyhow remained unsolved for all other BMC depths 7,8,9,10,11,12,15 during the competition. For the crafted track we used all 200 instances.

The scrambling strategies on the x-axis of these plots are ordered with respect to the best solver on the original instances in these tracks, which is "cadical" for the application track and "gluvc" for the crafted track. While in the crafted track "gluvc" clearly dominates as did "maplelcm" on the 2017 instances, the winner in the application track would vary depending on which scoring scheme and which scrambling strategy is applied, i.e., "maplelcm" would have won for "par2" scores with "`c010`" scrambling. The conclusion is that the same effects on performance can be observed for the SAT 2016 Competition benchmarks as for the SAT 2017 Competition benchmarks, except that for crafted instances scrambling has less influence.

Altogether we were running 7 solvers on 20 variants of $849 = 350 + 299 + 200$ benchmarks, which in total amounts to 16980 benchmarks and thus $118860 = 7 \cdot 16980$ individual runs, each with a time limit of 5000 seconds. This would take slightly less than 19 years of total CPU time if all runs would need to use the full time limit.

The actual runs required 12 years and 160 days of total CPU time: 4 years 237 days for the benchmarks of the main track of the SAT 2017 Competition, 4 years 11 days for the benchmarks of the application track and 3 years and 277 days for the benchmarks of the crafted track of the SAT 2016 Competition benchmarks.

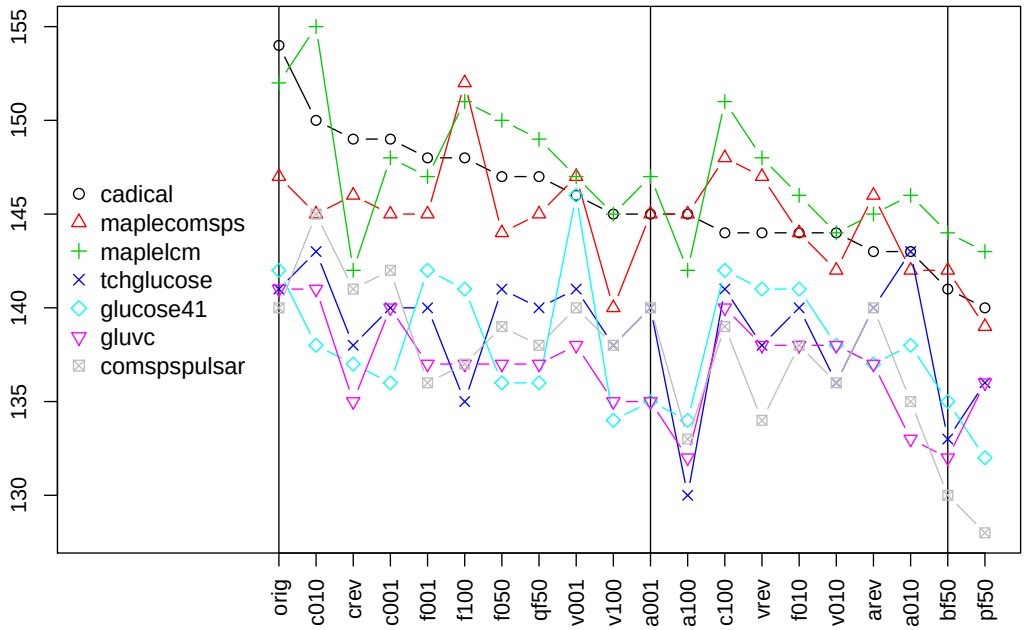

Figure 10: Solved instances (y-axis) from application track of the SAT 2016 Competition after applying different forms of scrambling (x-axis) – larger is better.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
cadical	1	1.15	0.37	1	1.95	0.83	1	4.95	1.67	1	1.15	0.37
maplecomsps	2	2.30	0.80	3	2.55	0.89	2	1.85	0.99	3	3.10	0.55
maplelcm	3	2.85	0.88	2	1.60	0.68	3	1.90	1.17	2	2.20	1.24
tchglucose	4	4.55	0.94	5	4.85	0.99	4	3.45	1.00	7	5.85	1.09
glucose41	5	5.30	1.42	4	5.30	1.42	5	5.35	1.57	5	4.65	1.35
gluvc	6	6.15	0.59	6	6.10	0.55	7	5.80	1.15	4	5.00	1.26
comspspulsar	7	5.70	1.30	7	5.65	1.23	6	4.70	1.72	6	6.05	0.76

Table 6: Original (o) and average (μ) ranking with standard deviation (σ) on benchmarks from the application track of the SAT 2016 Competition.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
cadical	1561368	1643180	37184	154	145.80	3.29	65	59.50	2.67	89	86.30	1.26
maplecomsps	1642508	1670404	28498	147	144.80	2.91	64	63.40	1.88	83	81.40	1.50
maplelcm	1644453	1681871	30163	152	147.10	3.48	64	63.05	2.33	88	84.05	2.68
tchglucose	1686618	1716486	28651	141	138.70	3.29	62	60.65	1.63	79	78.05	2.19
glucose41	1687616	1729213	30800	142	138.05	3.55	62	58.35	2.50	80	79.70	1.98
gluvc	1702788	1738847	27915	141	136.85	2.64	60	58.45	1.36	81	78.40	2.62
comspspulsar	1715156	1734357	34491	140	137.45	4.03	60	59.35	2.25	80	78.10	2.73

Table 7: Original (o) and average (μ) scores with standard deviation (σ) on benchmarks from the application track of the SAT 2016 Competition.

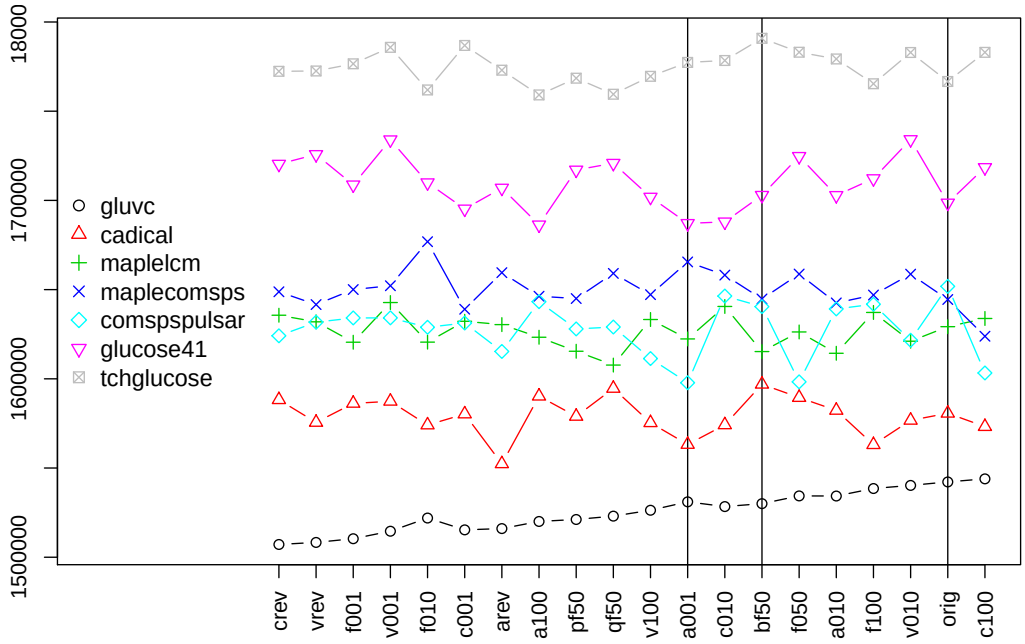

Figure 11: Par2 score (y-axis) for instances from crafted track of the SAT 2016 Competition after applying different forms of scrambling (x-axis) – smaller is better.

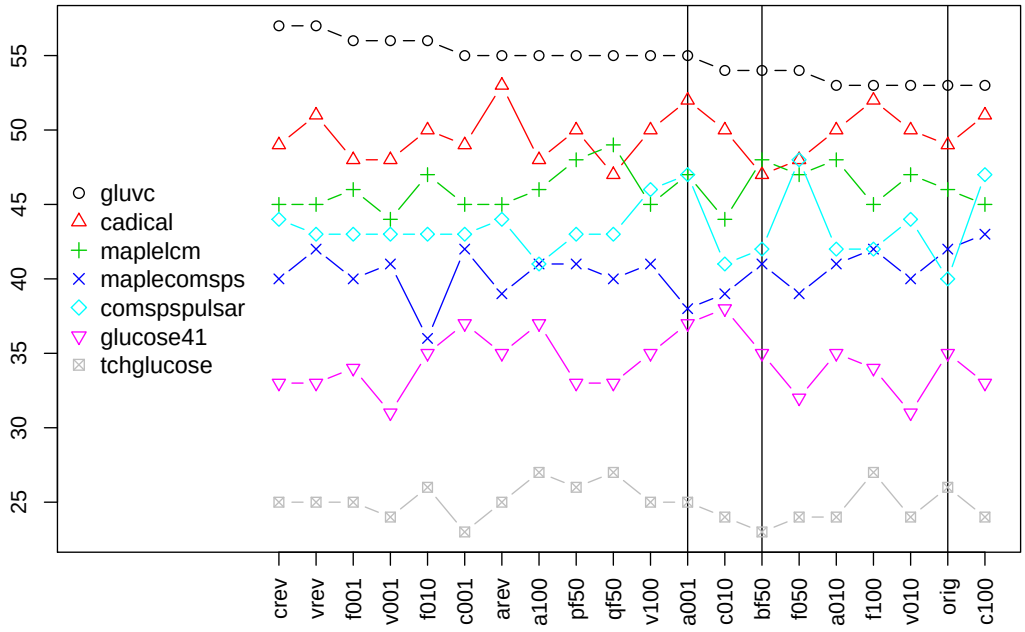

Figure 12: Solved instances (y-axis) from crafted track of the SAT 2016 Competition after applying different forms of scrambling (x-axis) – larger is better.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
gluvc	1	1.00	0.00	1	1.00	0.00	7	4.15	2.08	1	1.00	0.00
cadical	2	2.00	0.00	2	2.10	0.31	5	6.00	1.26	2	2.05	0.22
maplelcm	3	3.50	0.61	3	3.10	0.55	3	3.95	1.88	3	3.15	0.49
maplecomsps	4	4.90	0.31	4	4.95	0.22	1	2.50	1.67	5	5.00	0.00
comspspulsar	5	3.60	0.60	5	3.85	0.49	6	3.90	1.77	4	3.80	0.41
glucose41	6	6.00	0.00	6	6.00	0.00	4	4.15	1.69	6	6.00	0.00
tchglucose	7	7.00	0.00	7	7.00	0.00	2	3.35	2.03	7	7.00	0.00

Table 8: Original (o) and average (μ) ranking with standard deviation (σ) on benchmarks from the crafted track of the SAT 2016 Competition.

	par2			solved			sat			unsat		
	o	μ	σ	o	μ	σ	o	μ	σ	o	μ	σ
gluvc	1542210	1525377	11383	53	54.70	1.30	5	6.15	0.88	48	48.55	0.94
cadical	1580688	1579156	11125	49	49.60	1.67	5	5.10	0.85	44	44.50	1.43
maplelcm	1629226	1626691	9516	46	46.10	1.45	6	5.90	1.02	40	40.20	1.47
maplecomsps	1644454	1650459	11235	42	40.40	1.64	8	6.75	1.16	34	33.65	1.18
comspspulsar	1651826	1627593	15569	40	43.45	2.11	5	6.10	0.97	35	37.35	1.79
glucose41	1698562	1709790	14641	35	34.30	1.98	5	6.00	0.86	30	28.30	1.78
tchglucose	1766652	1774629	9319	26	24.95	1.23	7	6.40	0.94	19	18.55	0.69

Table 9: Original (o) and average (μ) scores with standard deviation (σ) on benchmarks from the crafted track of the SAT 2016 Competition.

Conclusion

We presented a range of scrambling techniques based on permuting variables and clauses, and flipping literals in order to determine the robustness of state-of-the-art SAT solvers. We observed that most scrambling strategies reduce performance of SAT solvers, but were surprised to see that some strategies actually improve performance. Researchers that claim that a new technique is useful should take this point into consideration. We also noticed that solvers are more robust on the SAT 2017 Competition benchmark suite than on the suite of SAT 2016 Competition. This could be a coincidence, but could also be explained as follows: most (possibly all) solvers have been optimized using the 2016 suite as part of the training set in contrast to the 2017 suite. As a consequence, solvers may be overfitted on the existing benchmarks.

As future work we want to explore more sophisticated ways of scrambling and compare external scrambling to internal scrambling within the solver through for instance randomizing the initial phase or the initial variable order, as well as using random decisions. Note, that our analysis of the source code and runs of the 7 considered solvers indicates than none of them used any kind of such randomization (was switched off by default). Only "maplelcm" should be considered non-deterministic since it switches to a different solving strategy after 2500 seconds.

Scrambling is only one technique that can be helpful to improve the quality of solver (or in general algorithm) evaluations. Based on our results we expect that the presented form of scrambling is mostly useful to enhance the evaluation in papers as they only use existing benchmarks. The quality of the results of SAT competitions is expected to gain more from a sophisticated selection of benchmarks than scrambling them. However, some of our light-weight scrambling strategies appear not harmful and may thus result in modest improvements.

Although the paper focusses on the use of scrambling to evaluate solver performance, there are other applications we want to explore. For example, we want to apply scrambling in the context of random sampling of satisfying assignments. Current uniform random sampling techniques are expensive, while the scrambling techniques discussed in this paper have only modest impact on performance. We want to understand whether there exist scrambling techniques that result in (near) uniform sampling. Another application is randomizing proofs of unsatisfiability via scrambling, which could be useful to randomize search for unsatisfiable cores.

References

[1] Gilles Audemard and Laurent Simon. Experimenting with small changes in conflict-driven clause learning algorithms. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings*, volume 5202 of *Lecture Notes in Computer Science*, pages 630–634. Springer, 2008.

[2] Gilles Audemard and Laurent Simon. Glucose and Syrup in the SAT'17. In Balyo et al. [5], pages 16–17.

[3] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo. SAT competition 2016: Recent developments. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the 31st AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 5061–5063. AAAI Press, 2017.

[4] Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors. *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, volume B-2016-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2016.

[5] Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors. *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2017.

[6] Daniel Le Berre and Laurent Simon. The essentials of the SAT 2003 competition. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2003.

[7] Daniel Le Berre and Laurent Simon. Fifty-five solvers in Vancouver: The SAT 2004 competition. In Holger H. Hoos and David G. Mitchell, editors, *SAT 2004, Vancouver, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 321–344. Springer, 2004.

[8] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Balyo et al. [5], pages 14–15.

[9] Jingchao Chen. Glu_vc: Hacking Glucose by Weighted Variable State Independent Decay Sum Branching Policy. In Balyo et al. [5], page 19.

[10] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[11] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[12] Marijn J. H. Heule. The DRAT format and DRAT-trim checker. CoRR, abs/1610.06229, 2016.

[13] Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS_LRB_VSIDS and MapleCOMSPS_CHB_VSIDS. In Balyo et al. [5], pages 20–21.

[14] Seongsoo Moon and Inaba Mary. bs_glucose, tch_glucose. In Balyo et al. [5], pages 24–25.

[15] Mladen Nikolic. Statistical methodology for comparison of SAT solvers. In Ofer Strichman and Stefan Szeider, editors, *SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of

Lecture Notes in Computer Science, pages 209–222. Springer, 2010.

[16] Chanseok Oh. COMiniSatPS Pulsar and GHackCOMSPS. In Balyo et al. [5], pages 12–13.

[17] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João Marques-Silva and Karem A. Sakallah, editors, *SAT 2007, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[18] Fan Xiao, Mao Luo, Chu-Min Li, Felip Manyà, and Zhipeng Lü. MapleLRB_LCM, Maple_LCM, Maple_LCM_Dist, MapleLRB_LCMoccRestart and Glucose-3.0+width in SAT Competition 2017. In Balyo et al. [5], pages 22–23.