Scrambling and Descrambling SMT-LIB Benchmarks

Tjark Weber

Uppsala University, Uppsala, Sweden
`tjark.weber@it.uu.se`

Abstract

Each year, the benchmarks for the SMT Competition are drawn from a known pool of benchmarks, the SMT Library. Competing solvers, rather than determine benchmark satisfiability from scratch, could thus cheat by simply looking up the correct answer for each benchmark in the library. To make this form of cheating more difficult, benchmarks in the SMT Competition are scrambled. We demonstrate that the current scrambling algorithm, which has been in use since 2011, is ineffective at obscuring the original benchmark. We propose an improved scrambling algorithm, and show that the problem of identifying the original benchmark under this improved algorithm is GI-complete.

1 Introduction

The International Satisfiability Modulo Theories Competition (SMT-COMP) is an annual competition between SMT solvers [6]. The first competition was held in 2005 as a satellite event of the International Conference on Computer-Aided Verification (CAV). In 2015, the 10th installment of SMT-COMP was part of the SMT Workshop, again affiliated with CAV. (In 2013, an evaluation was conducted instead of a competition [7].) The SMT-COMP series has been instrumental in encouraging adoption of the common SMT-LIB input format [3], which is now the de facto standard for the input language of SMT solvers, and has spurred substantial advances in solver technology. In this regard, it is comparable to other competitions in automated theorem proving, such as the CADE ATP System Competition (CASC) [19, 20] and the SAT Competition [12, 14].

The SMT Competition exercises solvers in several tracks and logic divisions. The precise number of tracks and divisions has changed over the years. In this paper, we focus on the competition's Main Track, where solvers are applied to individual benchmarks and expected to produce a single answer for each benchmark, either satisfiable (`sat`) or unsatisfiable (`unsat`).

Benchmarks for the Main Track (and indeed for all competition tracks) are drawn from the SMT Library [17], a large library of input problems written in the SMT-LIB language. New benchmarks are regularly added to the SMT Library; the number of benchmarks used in the competition has grown from a modest 1,360 in 2005 [2] to 154,238 in 2015 [16]. Benchmarks may be submitted by anyone, including solver developers. To reduce the bias that this might otherwise introduce into competition results, it is a staple rule of SMT-COMP to make all eligible benchmarks publicly available some time before the competition starts. Solver developers are explicitly encouraged to scrutinize the benchmarks, and to test and tune their solvers accordingly. Benchmarks whose status (i.e., `sat` or `unsat`) is unknown are not eligible for the competition.

The fact that benchmarks are known in advance has potential for abuse. One could create a "solver" that uses its input benchmark as a search key, and simply looks up and reports the benchmark's known status in the SMT Library. Such a solver should perform very well on all benchmarks in the library—much better than genuine solvers, which employ some (semi-) decision procedure for the benchmark's logic to determine satisfiability from scratch—but it could not solve any new benchmarks, and would thus be all but useless for practical applications.

```
(set-logic UFNIA)
(set-info :status unsat)
(declare-fun f (Int Int) Int)
(declare-fun x () Int)
(assert (forall ((y Int)) (< (f y y) y)))
(assert (> x 0))
(assert (> (f x x) (* 2 x)))
(check-sat)
(exit)
```

```
(set-logic UFNIA)
(declare-fun x2 () Int)
(declare-fun x1 (Int Int) Int)
(assert (< (* x2 2) (x1 x2 x2)))
(assert (> x2 0))
(assert (forall ((x3 Int)) (> x3 (x1 x3 x3))))
(check-sat)
(exit)
```

Figure 1: Original (left) and scrambled (right) SMT-LIB benchmark.

The rules of the SMT Competition explicitly disallow such solvers: "[S]olvers must not rely on previously determined identifying syntactic characteristics of competition benchmarks in testing satisfiability. Violation of this rule is considered cheating." [8]

To make this form of cheating more difficult, benchmarks in the competition are lightly scrambled before they are presented to solvers. Figure 1 shows an original benchmark on the left, and a scrambled version of the benchmark on the right. Note that benchmark scrambling has different goals than benchmark pre-processing [4, 9]: while both must preserve (un)satisfiability, pre-processing usually aims to simplify the benchmark in a way that reduces solving time. Scrambling, on the other hand, should ideally have little to no impact on solving time, so that is does not distort the competition results. We describe the benchmark scrambler that has been used at recent SMT Competitions in more detail in Section 2.

Scrambling prevents solvers from directly looking up their (scrambled) input benchmark in the SMT Library. However, depending on the scrambling algorithm used, it may still be possible to identify the original benchmark. In Section 3, we show that for the current scrambling algorithm, this can in fact be done with very little computational effort. The current scrambler is therefore ineffective at obscuring the original benchmark. We exploit this weakness in Section 4 by implementing a cheating SMT solver that outperforms the best solvers in the 2015 SMT Competition by two orders of magnitude. Moreover, in Section 5 we identify all benchmarks in the SMT Library that are scrambled versions of each other.

Motivated by this shortcoming of the current scrambling algorithm, we identify general requirements on scrambling algorithms in Section 6, and present an improved algorithm that makes it harder to identify the original benchmark. How much harder exactly? We show that the problem of identifying the original benchmark under the improved algorithm is as hard as the graph isomorphism problem (GI) [15], i.e., GI-complete. We evaluate an implementation of the improved algorithm on the entire SMT Library, and find that it is sufficiently efficient to be used in future SMT Competitions. Section 7 concludes.

2 Benchmark Scrambling

The current benchmark scrambler [11] was originally developed by Alberto Griggio. It has (with minor modifications) been used at every SMT Competition since 2011. To our knowledge, the high-level ideas behind its implementation have not been described before.

The scrambler is written in C++. It uses a Flex/Bison parser to transform SMT-LIB benchmarks into an abstract syntax tree. The tree is then printed again in concrete SMT-LIB syntax. Readers unfamiliar with the SMT-LIB language are referred to [3] for a comprehensive and authoritative description.

The following modifications are performed by the scrambler during parsing or printing:

1. Comments, **set−info** commands and other artifacts that have no logical effect (e.g., superfluous white-space, redundant parentheses) are removed.
2. Input names, in the order in which they are encountered during parsing, are replaced by names of the form x1, x2, ...
3. Variables bound by the same binder (let , forall , or exists) are shuffled.
4. Arguments to commutative operators (e.g., and, distinct , +) are shuffled.
5. Anti-symmetric operators (e.g., <, bvslt) are randomly replaced by their counterparts (e.g., >, bvsgt), with the arguments suitably swapped.
6. Consecutive declarations (i.e., **declare−fun** commands) are shuffled.
7. Consecutive assertions (i.e., **assert** commands) are shuffled.

Steps 4 and 5 are performed only if the resulting term is syntactically valid according to the benchmark's logic. Specifically, in difference logics, which impose substantial restrictions on permitted terms, some operators are exempt from these two transformations.

All shuffling and swapping, i.e., steps 3–7 above, is effected by pseudo-random choices. For instance, each occurrence of an anti-symmetric operator is flipped with probability approximately 0.5. Therefore, each benchmark can typically be scrambled in many different ways. The scrambler's actual output is determined by the seed value that is used to initialize the pseudo-random number generator.[1]

The scrambler supports further options, e.g., to deal with term annotations [3, §3.6]. These are not relevant for the Main Track of the competition.

3 Benchmark Normalization

Benchmark scrambling, as described in Section 2, is evidently a non-injective transformation. For instance, two benchmarks that differ only in comments or names will generate the same scrambled output (for any given seed). Therefore, it is not possible in general to undo the scrambling, i.e., to compute the original benchmark from the scrambler's output alone.

In the context of the SMT Competition, however, additional information is available: we can access the SMT Library, i.e., the collection of original benchmarks. This may still be insufficient to identify the original benchmark, as there could be several benchmarks in the SMT Library that would result in the same output when scrambled. (We quantify this phenomenon in Section 5.) But note that scrambling does not change the benchmark status: the scrambled output is (un)satisfiable if and only if the original benchmark is (un)satisfiable. Therefore, all possible original benchmarks are equisatisfiable, and for the purpose of cheating in the competition it is sufficient to identify any one of them.

To this end, we have implemented a *benchmark normalizer*.[2] The normalizer is applied to both original and scrambled benchmarks. It computes normal forms for these, such that the diagram in Figure 2 commutes.

We obtain the benchmark normalizer as a modification of the scrambler, by replacing all pseudo-random transformations (steps 3–7 in Section 2) with deterministic transformations that generate the same output regardless of the order of input arguments. In other words, where

[1] This seed value is computed after the SMT Competition's solver submission deadline each year, according to rules designed to make its value difficult to predict both for participants and competition organizers.

[2] Our implementations, also of the SMT solver described in Section 4 and of the improved scrambler described in Section 6, are available at `http://user.it.uu.se/~tjawe125/publications/weber16scrambling.html`. The evaluation data reported on in Sections 4-6 of this paper are available at the same URL.

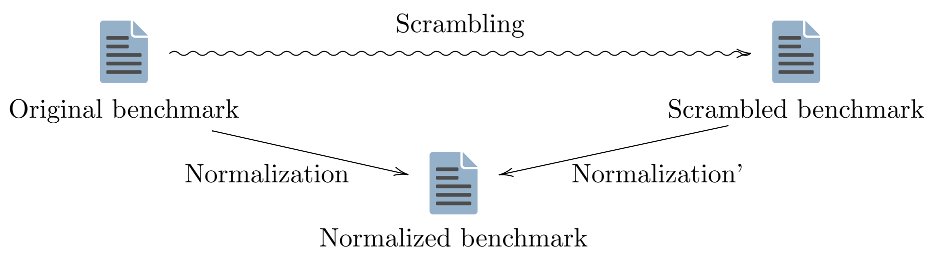


Figure 2: Benchmark normalization.

the scrambler shuffles, the normalizer sorts. Specifically, the normalizer performs the following operations:

1. Comments, **set−info** commands and other artifacts that have no logical effect (e.g., superfluous white-space, redundant parentheses) are removed.
2. When the normalizer is applied to an original benchmark, input names, in the order in which they are encountered during parsing, are replaced by names of the form x1, x2, ... When the normalizer is applied to a scrambled benchmark, input names are retained.
3. Variables bound by the same binder (let , forall , or exists) are sorted.
4. Arguments to commutative operators (e.g., and, distinct , +) are sorted.
5. For each pair of anti-symmetric operators (e.g., $<$ and $>$), we have chosen a canonical representation. All occurrences of the non-canonical operator are eliminated.
6. Consecutive declarations (i.e., **declare−fun** commands) are sorted.
7. Consecutive assertions (i.e., **assert** commands) are sorted.

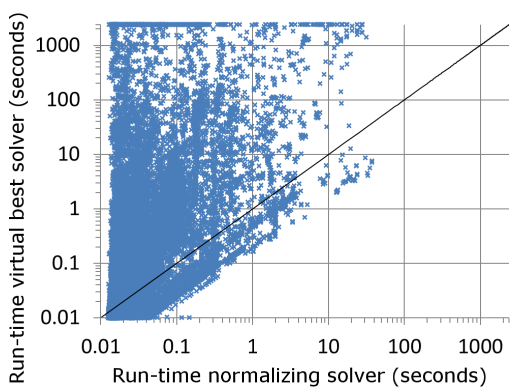
Sorting is performed with respect to a fixed total order on nodes of the abstract syntax tree. We sort nodes according to the symbol that they contain, and nodes with equal symbols according to a lexicographic ordering of their children. Any other total order could be used instead.

Note that the normalizer (just like the scrambler) applies many of these operations already during parsing, in a bottom-up fashion. Consequently, subterms are already in normal form when they are considered as constituents of larger terms. Therefore, flipping anti-symmetric operators (step 5) does not interfere with the sorting of arguments to commutative operators (step 4), regardless of the total order used.

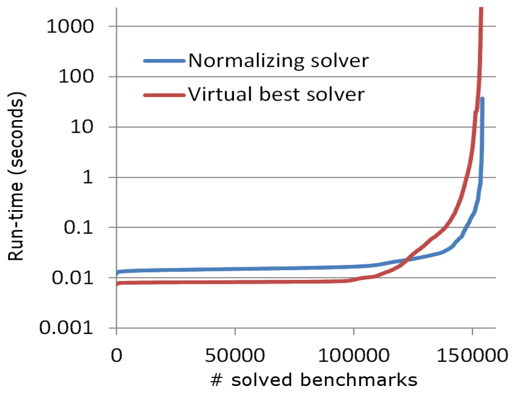
4 The World's Fastest SMT Solver

To evaluate the performance of benchmark normalization and its feasibility in the context of the SMT Competition, we have implemented a cheating SMT solver. First, we normalized all 154,238 benchmarks in the SMT Library that were used in the Main Track of the 2015 SMT Competition. For each normal form we computed its SHA-512 hash digest (there were no hash collisions), and used this to prime our solver with a map from digests to benchmark status. Note that this computation can be done offline, i.e., before the start of the competition.

We then applied our solver to all Main Track benchmarks, using the same StarExec [18] execution environment that was used for the 2015 SMT Competition. As in the competition, evaluation machines were equipped with Intel Xeon CPUs (E5-2609, 10 MB cache, 2.4 GHz),



(a) Run-time comparison for each benchmark.



(b) Run-times plotted against the number of benchmarks solved.

Figure 3: Performance of the normalizing solver on all 154,238 Main Track benchmarks of the 2015 SMT Competition, compared to a *virtual best solver* comprised of all solvers that participated in the competition.

memory was capped at 61,440 MB, and a time limit of 2,400 seconds per benchmark was imposed. Benchmarks were scrambled with the competition scrambler, using the same seed value (53689572) that was used in the competition, before they were presented to our solver. The choice of seed value is largely irrelevant. The seed is only passed to the competition scrambler—but not to our solver—and any other value should lead to similar evaluation results.

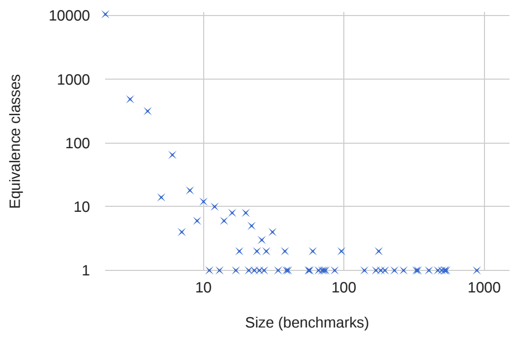
Our solver normalizes each scrambled benchmark (retaining input names), computes the SHA-512 digest of the normal form, and uses this to look up the benchmark's status in the pre-computed map.

Figure 3 shows the solver's performance, and compares it to that of a *virtual best solver*, which is obtained by using, for each benchmark, the best performance of any solver that participated in the 2015 SMT Competition. While the virtual best solver outperforms our cheating solver on some (mostly very easy) benchmarks, our solver gets ahead of the virtual best solver after 0.02 seconds. It returns a correct result for every competition benchmark; in fact, it takes at most 37 seconds to solve each benchmark. In contrast, the virtual best solver times out on 482 benchmarks. The total run-time of our solver on all competition benchmarks is 7,729 seconds—approximately a 223-fold speedup over the virtual best solver, which ran for 1,721,874 seconds total (including timeouts).

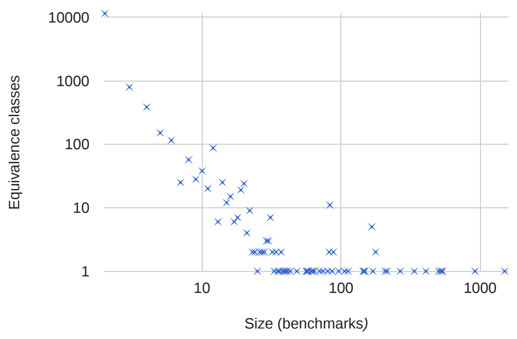
We conclude that the scrambling algorithm that has been in use at SMT Competitions since 2011 does not create significant computational obstacles to identifying the original benchmark. Even for the largest benchmarks used in the competition, normalization is feasible, and is usually orders of magnitude faster than solving the benchmark from scratch.

5 Benchmark Similarities in the SMT Library

All Main Track benchmarks contain exactly one **check−sat** command. Such benchmarks are called *non-incremental.* The SMT Library currently contains 196,375 non-incremental benchmarks. (Of these, 30,717 were not eligible for the SMT Competition in 2015 because their status was unknown, or because of syntactic restrictions. We include these benchmarks in our



(a) Identical benchmarks in SMT-LIB.



(b) Similar benchmarks in SMT-LIB.

Figure 4: The number of equivalence classes of non-incremental SMT-LIB benchmarks (vertical axis), plotted against the number of benchmarks contained in the class (horizontal axis). The equivalence relation used is syntactic identity in subfigure a, and *similarity* in subfigure b. Equivalence classes of size 1 (of which there are 163,810 for syntactic identity, and 152,227 for similarity) are elided.

analysis in this section.)

By computing and comparing the SHA-512 digests of all non-incremental benchmarks, we find that 21,084 of these benchmarks are duplicates. Some benchmarks have been copied many times; the front-runner appears 885 times under different benchmark names. Figure 4a shows in more detail how the SMT Library can be partitioned into equivalence classes of syntactically equal benchmarks, according to the number of times that each benchmark appears in the library.

The normalizer that was presented in Section 3 allows us to repeat this investigation with a coarser equivalence relation in place of syntactic identity: namely, we call two benchmarks *similar* if they have the same normal form. Under this equivalence relation, which would consider all scrambled versions of a benchmark similar to the original, we find that 30,799 non-incremental benchmarks are duplicates. There are up to 1,499 similar versions of a single benchmark. Figure 4b shows the partition of the SMT Library into equivalence classes for this notion of similarity.

Interestingly, there are 119 non-incremental benchmarks in the SMT Library whose status is currently unknown, but that are similar to benchmarks with known status (and thus must be equisatisfiable). This suggests that benchmark normalization may have some legitimate use in SMT solving after all.

We suggest that duplicate benchmarks be removed from the SMT Library. To avoid giving undue emphasis to specific benchmarks, usually only one version of similar benchmarks should be retained. Moreover, we suggest that new benchmarks should be added to the library only after their dissimilarity from existing benchmarks has been established.

6 A GI-Complete Scrambling Algorithm

Is there a better scrambling algorithm, which is not as easy to sidestep? Before we can answer this question, we identify the following requirements on a good scrambling algorithm.

1. *The algorithm must not affect satisfiability.* Rationale: This fundamental correctness property ensures that the status of the scrambled benchmark is known. It is also necessary to argue that the scrambled output is a variation of the original problem.

2. *The algorithm must be efficient.* Rationale: Scrambling must not become a bottleneck in the SMT Competition. The largest benchmarks in the SMT Library have a size of several gigabytes. We found that even algorithms whose complexity was quadratic in the size of the benchmark were too slow to be useful in practice.
3. *Ideally, the algorithm should not affect solving times.* Rationale: Any impact on solving times would potentially distort the competition results. Unfortunately, this property cannot be fully guaranteed in practice—SMT solvers use complex heuristics, where even seemingly innocuous changes like renaming of variables may have tremendous impact [5]. However, the algorithm should certainly not alter the benchmark size or structure in any significant way.
4. *Given two benchmarks, it should be hard to decide without additional information (such as the seed used for scrambling) whether one is a scrambled version of the other.* Rationale: This property ensures that scrambling makes it difficult to cheat in the SMT Competition, i.e., that given a scrambled benchmark, one cannot easily identify the corresponding original benchmark in the SMT Library.

A good scrambling algorithm, therefore, is akin to a *one-way function* (i.e., a function that is easy to compute on every input, but hard to invert given the image of a random input) that preserves benchmark satisfiability, size and structure.

The existence of one-way functions is a famous open problem [10]. The scrambling algorithm that has been in use at SMT Competitions since 2011 meets the first three requirements above, but—as our evaluation in Section 4 demonstrated—falls short of the fourth.

To improve on this algorithm, we observe that the normalization algorithm presented in Section 3 crucially relies on the fact that the replacement of input names with names of the form x1, x2, ... is entirely predictable, as it only depends on the order in which input names appear in the original benchmark. We propose to amend the scrambling algorithm that was described in Section 2 with an additional step as follows:

2a. A (pseudo-)random permutation π is applied to all names x1, x2, ... that occur in the benchmark, replacing each name xi with $\pi(\mathsf{x}i)$.

With this addition, sorting a scrambled benchmark with respect to a fixed term order is no longer sufficient to normalize the benchmark. The normalizer would also have to replace scrambled names xi with their original names $\pi^{-1}(\mathsf{x}i)$. But π (and the seed from which π was derived) is not known to the normalizer, and it cannot be reconstructed easily, since the other steps of the scrambling algorithm obfuscate the connection between original and scrambled names. In fact, we can prove the following theorem.

Theorem 1. *For the scrambling algorithm from Section 2, amended with a name-permuting step as detailed above, the problem of determining whether (there exists a seed such that) two benchmarks are scrambled versions of each other is GI-complete.*

Recall that GI is the class of problems that have a polynomial-time Turing reduction to the *graph isomorphism problem* [15], i.e., the problem of determining whether two finite graphs are isomorphic. This problem is of great interest in complexity theory: it lies in NP, but is not known to be NP-complete, nor is it known to be solvable in polynomial time. The following proof of Theorem 1 incorporates ideas by Tsuyoshi Ito [13].

Proof. To reduce the problem of determining whether two benchmarks are scrambled versions of each other to graph isomorphism, consider an SMT-LIB benchmark S. Without loss of generality we may assume that S consists of a single block of declarations followed by a single block of assertions. (Otherwise, we apply the following construction separately to each block.) As in step 5 of the normalization algorithm (Section 3), replace anti-symmetric operators in S with their canonical representation. Then construct a vertex-labeled graph $G(S)$ as follows.

1. For each input name x that occurs in S, the graph has a vertex v_x labeled as “name.”
2. For each assertion ϕ_i in S, the graph contains a syntax tree for ϕ_i. Arguments to non-commutative operators in the syntax tree are suitably labeled (e.g., with non-negative integers) to indicate their order. For commutative operators, no such labeling is present.
3. Syntax trees do *not* contain input names. Instead, for each occurrence of an input name x in ϕ_i, there is an edge connecting the appropriate parent in the syntax tree to v_x.

Two benchmarks S and T are scrambled versions of each other if and only if they can be made equal by renaming, reordering arguments to commutative operators, and reordering assertions. It is not difficult to see that this is the case if and only if there is an isomorphism between $G(S)$ and $G(T)$ that preserves labels. But the isomorphism problem of vertex-labeled graphs is known to be equivalent to the isomorphism problem of unlabeled graphs [21]. This proves that the above problem is in GI.

To reduce graph isomorphism to the problem of determining whether two benchmarks are scrambled versions of each other, given a graph $G = (V, E)$, construct an SMT-LIB benchmark $B(G)$ as follows. For each vertex $v \in V$, $B(G)$ contains a declaration (**declare−fun** v () Bool). For each edge $e = \{v_1, v_2\} \in E$, $B(G)$ contains an assertion (**assert** $(= v_1\ v_2)$). Then again it is not difficult to see that two graphs G and H are isomorphic if and only if $B(G)$ and $B(H)$ are scrambled versions of each other. Therefore, the above problem is also GI-hard. □

We have implemented this addition to the scrambling algorithm, and evaluated the amended algorithm by scrambling all 196,375 non-incremental benchmarks in the SMT Library (using the StarExec execution environment and seed value described in Section 4). Across the entire SMT Library, the (unoptimized) amended implementation incurs 28% overhead when compared to the current scrambler. The maximal scrambling time for a single benchmark increases from 1,289 seconds to 1,786 seconds. We conclude that the amended algorithm, albeit clearly slower than the current scrambler, is sufficiently efficient to be used in future SMT Competitions.

7 Conclusion

We have shown that the scrambling algorithm that has been in use to obscure SMT-LIB benchmarks in SMT Competitions since 2011 is ineffective. A cheating solver that identifies the original benchmark and looks up the benchmark’s correct answer in the SMT Library outperforms the best solvers in the 2015 SMT Competition by two orders of magnitude. This also helped us to identify numerous duplicate benchmarks in the SMT Library, including 119 benchmarks whose status was previously unknown.

Despite the relative technical ease with which the current scrambling algorithm can be sidestepped, we have no reason to believe that this form of cheating has actually occurred at past SMT Competitions. Winning solvers receive little beyond recognition by the research community; consequently, there are strong social disincentives to submitting a cheating solver. Moreover, most competing solvers—including those that performed best—are open source, allowing their implementations to be scrutinized.

We have proposed and implemented an improved scrambling algorithm. Under this algorithm, identifying the original benchmark is GI-complete. Our implementation of the improved algorithm is (on average) only 28% slower than the current scrambler on the SMT Library, and we expect it to be used for the 2016 SMT Competition. Our results might also prompt the organizers of other competitions that use previously known benchmarks, such as CASC and SAT, to revisit the issue of benchmark scrambling.

At this point, at least two open questions remain. First, it is unclear whether the improved scrambling algorithm makes it practically infeasible to recognize the original benchmark. Although no polynomial-time algorithm for solving GI is known, for many—if not all—existing benchmarks in the SMT Library this may well be sufficiently easy. Recent work by Babai [1] claims to show that GI is quasi-polynomial. Even with the improved scrambling algorithm, the SMT Competition may thus have to rely on social disincentives and scrutiny more than on technical measures to prevent this form of cheating.

Second, is there a scrambling algorithm that meets the requirements of Section 6 and—for theoretical or practical reasons—is better than the algorithm proposed here? In the terminology of [5], we have advocated moving from an “identity” scrambling algorithm, which only shuffles declarations, assertions and terms within assertions, to a “permutation” algorithm, which additionally shuffles names. Further transformations are conceivable, e.g., complementing (negating) Boolean or numerical symbols. We leave these questions for future work.

8 Acknowledgments

Sylvain Conchon and David Déharbe organized SMT-COMP 2015 together with the author of this paper.

References

[1] László Babai. Graph isomorphism in quasipolynomial time. *CoRR*, abs/1512.03547, 2015.

[2] Clark Barrett, Morgan Deters, Leonardo Mendonça de Moura, Albert Oliveras, and Aaron Stump. 6 years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf. Retrieved April 13, 2016.

[4] Richard Bonichon, David Déharbe, Pablo Dobal, and Cláudia Tavares. SMTpp: preprocessors and analyzers for SMT-LIB. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*, 2015. To appear.

[5] Franc Brglez, Xiao Yu Li, and Matthias F. Stallmann. On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Annals of Mathematics and Artificial Intelligence*, 43(1):1–34, 2004.

[6] David R. Cok, David Déharbe, and Tjark Weber. The 2014 SMT competition. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:207–242, 2014.

[7] David R. Cok, Aaron Stump, and Tjark Weber. The 2013 evaluation of SMT-COMP and SMT-LIB. *Journal of Automated Reasoning*, 55(1):61–90, 2015.

[8] Sylvain Conchon, David Déharbe, and Tjark Weber. 10th International Satisfiability Modulo Theories Competition (SMT-COMP 2015): Rules and procedures. http://smtcomp.sourceforge.net/2015/rules15.pdf. Retrieved April 13, 2016.

[9] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 222–236. Springer, 2011.

[10] Oded Goldreich. *Foundations of Cryptography: Volume 1, Basic Tools.* Cambridge University Press, 2007.

[11] Alberto Griggio. SMT-COMP 2011 benchmark scrambler. http://smtcomp.sourceforge.net/2015/smtcomp2015_scrambler.tar.gz. Retrieved April 13, 2016.

[12] Marijn Heule and Torsten Schaub. What's hot in the SAT and ASP competitions. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.*, pages 4322–4323. AAAI Press, 2015.

[13] Tsuyoshi Ito. What's the complexity of recognizing equivalence for the following relation? (Answer), 2014. http://cstheory.stackexchange.com/questions/27287/whats-the-complexity-of-recognizing-equivalence-for-the-following-relation. Retrieved April 13, 2016.

[14] Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The international SAT solver competitions. *AI Magazine*, 33(1), 2012.

[15] J. Kobler, U. Schöning, and Jacobo Toran. *The Graph Isomorphism Problem: Its Structural Complexity.* Progress in Theoretical Computer Science. Birkhäuser, Boston, 1993.

[16] SMT-COMP 2015 — benchmarks. http://smtcomp.sourceforge.net/2015/benchmarks.shtml. Retrieved April 13, 2016.

[17] SMT-LIB The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu/benchmarks.shtml. Retrieved April 13, 2016.

[18] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 367–373. Springer, 2014.

[19] G. Sutcliffe. The 7th IJCAR automated theorem proving system competition - CASC-J7. *AI Communications*, 28(4):683–692, 2015.

[20] G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.

[21] V. N. Zemlyachenko, N. M. Korneenko, and R. I. Tyshkevich. Graph isomorphism problem. *Journal of Soviet Mathematics*, 29(4):1426–1481, 1985.