

Scrambling and Descrambling SMT-LIB Benchmarks

Tjark Weber

Uppsala University, Sweden

SMT 2016
Coimbra, Portugal

Motivation

The benchmarks used in the SMT Competition are known in advance.

Competing solvers could **cheat** by simply looking up the correct answer for each benchmark in the SMT Library.

To make this form of cheating more difficult, benchmarks in the competition are lightly **scrambled**.

Scrambling: Example

```
(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)
```

ORIGINAL
BENCHMARK

Scrambling: Example

```
(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)
```

ORIGINAL
BENCHMARK

SCRAMBLED
BENCHMARK

The Benchmark Scrambler

The benchmark scrambler **parses** SMT-LIB benchmarks into an abstract syntax tree, which is then **printed** again in concrete SMT-LIB syntax.

- Originally developed by Alberto Griggio
- Written in C++ (\approx 1,000 lines of code)
- Based on a Flex/Bison parser (\approx 900 lines) for the SMT-LIB language
- Used (with minor modifications) at every SMT-COMP since 2011

The (Old) Scrambling Algorithm

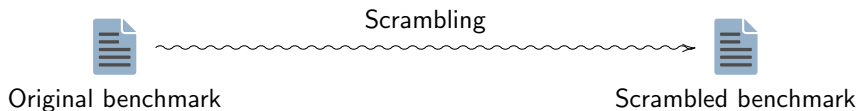
- 1 Comments and other artifacts that have no logical effect are removed.
- 2 Input names, in the order in which they are encountered during parsing, are replaced by names of the form x_1, x_2, \dots .
- 3 Variables bound by the same binder (e.g., `let`, `forall`) are shuffled.
- 4 Arguments to commutative operators (e.g., `and`, `+`) are shuffled.
- 5 Anti-symmetric operators (e.g., `<`, `bvslt`) are randomly replaced by their counterparts (e.g., `>`, `bvsgt`).
- 6 Consecutive declarations are shuffled.
- 7 Consecutive assertions are shuffled.

All pseudo-random choices depend on a seed value that is not known to competition solvers.

Benchmark Normalization

Since scrambling loses information (e.g., input names), the original benchmark cannot be restored from the scrambled benchmark alone.

However, how difficult is it to identify some original benchmark(s) in the SMT Library that could have resulted in the scrambled output?

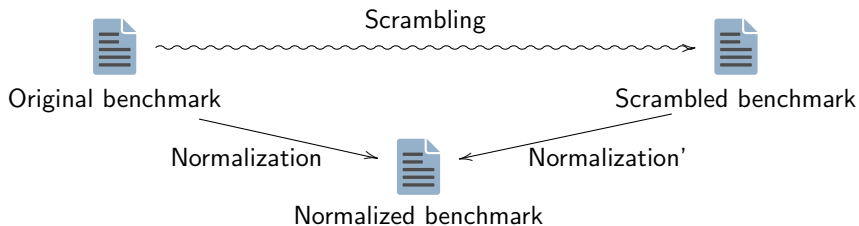


Benchmark Normalization

Since scrambling loses information (e.g., input names), the original benchmark cannot be restored from the scrambled benchmark alone.

However, how difficult is it to identify some original benchmark(s) in the SMT Library that could have resulted in the scrambled output?

This turns out to be computationally easy. We use a [normalization](#) algorithm:



The Normalization Algorithm

- 1 Comments and other artifacts that have no logical effect are removed.
- 2 For **original benchmarks**, input names, in the order in which they are encountered during parsing, are replaced by names of the form x_1, x_2, \dots . For **scrambled benchmarks**, input names are retained.
- 3 Variables bound by the same binder (e.g., `let`, `forall`) are **sorted**.
- 4 Arguments to commutative operators (e.g., `and`, `+`) are **sorted**.
- 5 Anti-symmetric operators (e.g., `<`, `bvslt`) are replaced by a **canonical representation**.
- 6 Consecutive declarations are **sorted**.
- 7 Consecutive assertions are **sorted**.

Where the scrambler shuffles, the normalizer sorts.

The World's Fastest SMT Solver

Our normalization algorithm allows us to build a cheating SMT solver.

Before the competition:

- 1 Normalize all 154,238 benchmarks used in the Main Track of SMT-COMP 2015.
- 2 For each normal form, compute its SHA-512 hash digest. Create a map from digests to benchmark status.

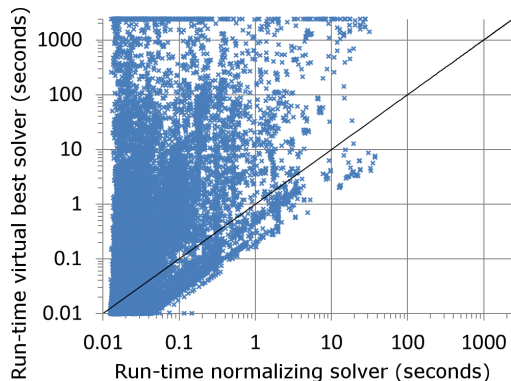
During the competition, for each scrambled benchmark:

- 1 Normalize the benchmark (retaining input names).
- 2 Compute the SHA-512 digest of the normal form.
- 3 Use this to look up the benchmark's status in the pre-computed map.

The World's Fastest SMT Solver: Performance

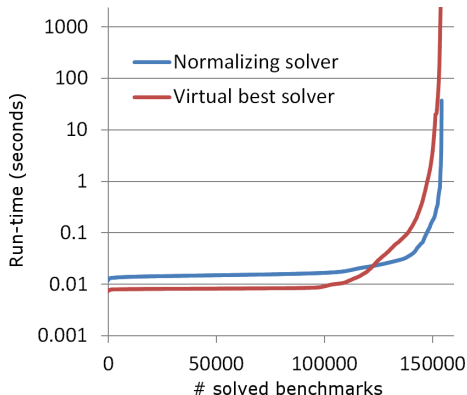
We compare the performance of our normalizing solver to the performance of a **virtual best solver** obtained by using, for each benchmark, the best performance of any solver that participated in SMT-COMP 2015.

Run-time comparison for each benchmark:



The World's Fastest SMT Solver: Performance (cont.)

Run-times plotted against the number of benchmarks solved:



Our normalizing solver **solves every benchmark** and is (on average) **223 times faster** than the virtual best solver.

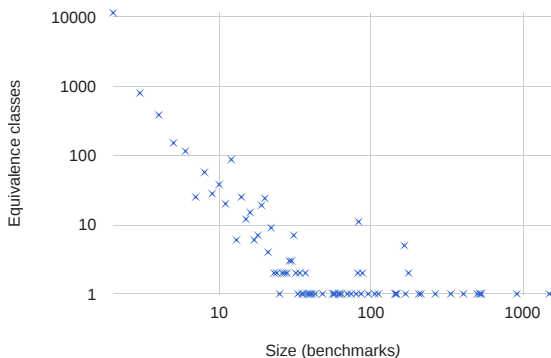
Benchmark Similarities in the SMT Library

Our normalization algorithm allows us to identify **similar benchmarks** in the SMT Library.

There are 196,375 non-incremental benchmarks in the 2015 release of the SMT Library.

We call two benchmarks similar if they have the same normal form.

Benchmark Similarities in the SMT Library: Findings



- 30,799 benchmarks (16%) are duplicates wrt. similarity.
- Up to 1,499 similar versions of a single benchmark.
- 119 benchmarks with unknown status are similar (and thus equisatisfiable) to benchmarks with known status.

Requirements on a Good Scrambling Algorithm

- 1 Must not affect satisfiability.
- 2 Must be efficient.
- 3 Should (ideally) not affect solving times.
- 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.

Requirements on a Good Scrambling Algorithm

- 1 Must not affect satisfiability.
- 2 Must be efficient.
- 3 Should (ideally) not affect solving times.
- 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.

The old scrambling algorithm meets (1)-(3), but falls short of (4).

Observation: Our normalization algorithm crucially relies on the fact that the replacement of input names with names of the form x_1, x_2, \dots is entirely predictable.

A New Scrambling Algorithm

- 1 Comments and other artifacts that have no logical effect are removed.
- 2 Input names, in the order in which they are encountered during parsing, are replaced by names of the form x_1, x_2, \dots
- 3 A random permutation π is applied to all names, replacing each name x_i with $\pi(x_i)$.
- 4 Variables bound by the same binder (e.g., `let`, `forall`) are shuffled.
- 5 Arguments to commutative operators (e.g., `and`, `+`) are shuffled.
- 6 Anti-symmetric operators (e.g., `<`, `bvslt`) are randomly replaced by their counterparts (e.g., `>`, `bvsgt`).
- 7 Consecutive declarations are shuffled.
- 8 Consecutive assertions are shuffled.


The New Scrambling Algorithm is GI-Complete

Theorem

For the new scrambling algorithm, the problem of determining whether two benchmarks are scrambled versions of each other is GI-complete.

Proof of GI-hardness:

Given a graph $G = (V, E)$, construct a corresponding SMT-LIB benchmark $B(G)$ as follows:

$v \in V$  **(declare-fun** v **() Bool)**
 $\{v1, v2\} \in E$ **(assert (= v1 v2))**

Now two graphs G and H are isomorphic if and only if $B(G)$ and $B(H)$ are scrambled versions of each other.

Conclusions

The scrambling algorithm used at SMT-COMP since 2011 is **ineffective** at obscuring the original benchmark. However, we have no reason to believe that cheating has occurred at past competitions.

Our improved scrambling algorithm renders the problem of identifying the original benchmark **GI-complete**. This algorithm has now been used at SMT-COMP 2016.

Nonetheless, the competition may have to rely on **social disincentives** and **scrutiny** more than on technical measures to prevent this form of cheating.

Is there an even better scrambling algorithm?