Efficient Identification of Isomorphic SAT Instances

Ashlin Iser
Karlsruhe Institute of Technology, Germany

Frederick Gehm
Karlsruhe Institute of Technology, Germany

Abstract

Many SAT benchmark datasets contain structurally identical instances arising from repeated shuffling, generators producing identical formulas under different seeds, or duplicate encodings from different tools. We present an efficient, open-source, isomorphism-invariant hashing algorithm for SAT instances, based on Weisfeiler–Leman (WL) label refinement. Each instance is represented as a bipartite clause–literal graph, and iterative label refinement computes a canonical signature, with instances having identical signatures treated as isomorphic. When integrated into our benchmark toolset Global Benchmark Database (GBD), the method substantially reduces false positives from naive degree-sequence hashing with minimal overhead.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Design and analysis of algorithms; Information systems → Information integration

Keywords and phrases Isomorphism, Benchmarking, Boolean Satisfiability

Digital Object Identifier 10.4230/LIPIcs.SAT.2026.35

Category Tool Paper

Supplementary Material *Software*: `https://github.com/udopia/gbdc` [8]
archived at `swh:1:dir:2553a0109d95559020fb87331f2d49502d4c4c67`

Acknowledgements We thank Timon Passlick for his BA thesis on early explorations of fast isomorphism detection.

1 Introduction

Boolean satisfiability (SAT), the canonical NP-hard problem, has been a central focus of research in theoretical computer science and artificial intelligence for decades. Significant advancements in hardware and software verification, electronic design automation, and combinatorial optimization result from progress in efficient SAT solvers. Benchmarking is essential for evaluating and comparing solver performance.

However, isomorphic instances inflate benchmark datasets and skew empirical results, making it difficult to assemble diverse, non-redundant benchmark sets. Isomorphic instances arise from various sources: scrambled instances in competition benchmark sets [4], structurally identical encodings from various applications or parameter settings [14], and generators producing identical formulas under different seeds [16].

Instances of the SAT problem are typically presented in Conjunctive Normal Form (CNF), in which formulas are given as a conjunction of clauses, where each clause is a disjunction of literals. Isomorphic instances are those that are structurally identical, meaning they can be transformed into one another through renaming of variables, reordering of clauses and literals within clauses, and flipping of literal polarities. Current detection methods typically employ basic heuristics that produce many false positives [7], or graph isomorphism algorithms that prove computationally prohibitive for large datasets [1].

We present an efficient approach for identifying and eliminating isomorphic instances from SAT benchmark collections using isomorphism-invariant hashing based on one-dimensional Weisfeiler-Lehman (WL) refinement [15]. We empirically compare implementation variants and characterize isomorphism prevalence across SAT competition benchmarks. Integration into the Global Benchmark Database (GBD) [10] enables efficient detection and improves empirical evaluation reliability.

The remainder of this paper is structured as follows: In Section 2, we introduce key concepts and review related work. In Section 3, we describe the implementation of our isomorphism-invariant hashing tool. In Section 4, we present an empirical evaluation and analyse isomorphism prevalence in existing benchmark datasets. Finally, we conclude with Section 5, where we summarise our findings and suggest directions for future research.

2 Background

Given a finite set of variables $V = \{v_1, v_2, \ldots, v_n\}$, a Boolean formula F in Conjunctive Normal Form (CNF) is a conjunction of clauses $F = \{C_1, C_2, \ldots, C_m\}$, where each clause C_i is a disjunction of literals $C_i = \{l_1, l_2, \ldots, l_k\}$ with literals drawn from $L = V \cup \{\neg v \mid v \in V\}$. The complement function over literals is defined such that $\bar{l} = \neg v$ if $l = v$ and $\bar{l} = v$ if $l = \neg v$. In the following, we denote the set of variables of a formula F as $\mathsf{vars}(\mathsf{F})$ and its set of literals as $\mathsf{lits}(\mathsf{F})$.

Two CNF formulas are isomorphic if one can be transformed into the other by renaming variables, inverting polarities, and reordering clauses and literals within clauses. Formally, two CNF formulas F and F' are isomorphic if there exists a flipping function $\alpha : V \to \{0, 1\}$ inducing a bijection between literals $\phi_\alpha : L \to L$ such that $F' = \big\{\{\phi_\alpha(l) \mid l \in C\} \mid C \in F\big\}$, where

$$\phi_\alpha(l) := \begin{cases} \bar{l} & \text{if } \alpha(v) = 1 \\ l & \text{otherwise.} \end{cases}$$

CNF formula isomorphism reduces to hypergraph isomorphism. A CNF formula F can be represented as a hypergraph $H = (V_H, E_H)$, with vertices V_H corresponding to literals L and hyperedges E_H corresponding to clauses $C \in F$. The literal complement function induces a bijection $\mathsf{m} : V_H \to V_H$ with $\mathsf{m}(l) = \bar{l}$.

Hypergraph isomorphism can be polynomially reduced to graph isomorphism. Given a hypergraph H, we construct the corresponding bipartite graph $G = (V_L \cup V_C, E_G)$, where V_L represents the literals and V_C represents the hyperedges. The edges E_G connect the literals to their respective hyperedges. The graph isomorphism problem involves determining whether two graphs are structurally identical, i.e. whether a bijection exists between their vertex sets that preserves adjacency. We stipulate that this bijection must respect the literal complement function, ensuring symmetric mapping of literals and their complements.

Related Work

The graph isomorphism problem is noteworthy due to its unresolved complexity status: it belongs to the NP class, yet it is neither known to be NP-complete nor to be solvable in all cases in polynomial time. Recent advances, such as Babai's quasi-polynomial time algorithm, have improved our understanding, but the general case remains open [2].

Thanks to its practical efficiency, the label refinement algorithm (also known as the Weisfeiler-Lehman algorithm) is widely used to test for graph isomorphism [18]. It provides an over-approximation of hypergraph isomorphism and has been shown to be a valid test for almost all graphs [3]. Graphs that cannot be distinguished by this algorithm or its higher-dimensional variants have been characterised in [5].

Algorithm 1 IsoHash2.

Input: CNF formula F and maximum iterations $t_{\max}$
Output: IsoHash2

1 $K_0(l) \leftarrow 1 \quad \forall l \in \mathsf{lits}(\mathsf{F})$
2 $d \leftarrow 0$, $d' \leftarrow 1$
3 **for** ($t \leftarrow 0$; $d \neq d'$ and $t < t_{\max}$; $t = t + 1$)
4 $\quad A[l] \leftarrow 0 \quad \forall l \in \mathsf{lits}(\mathsf{F})$
5 $\quad$ `/* Aggregate Clause Labels */`
6 $\quad$ **foreach** $C \in F$ **do**
7 $\quad\quad X_C \leftarrow \mathsf{mix}\big(\{K_t(l) \mid l \in C\}\big)$
8 $\quad\quad$ **foreach** $l \in C$ **do** $A[l] \leftarrow A[l] + X_C$
9 $\quad$ `/* Update Literal Labels */`
10 $\quad$ **foreach** $l \in \mathsf{lits}(\mathsf{F})$ **do**
11 $\quad\quad K_{t+1}(l) \leftarrow \mathsf{combine}\big(A[l], K_t(l), K_t(\bar{l})\big)$
12 $\quad K_t \leftarrow K_{t+1}$
13 $\quad$ `/* Stabilization Check */`
14 $\quad d' \leftarrow d$
15 $\quad d \leftarrow \left|\left\{\big(K_{t+1}(v), K_{t+1}(\neg v)\big) \mid v \in \mathsf{vars}(\mathsf{F})\right\}\right|$
16 **return** $\mathsf{hash}\Big(\mathsf{sort}\Big(\big\{\mathsf{canon}(K_t(v), K_t(\neg v)) \mid v \in \mathsf{vars}(\mathsf{F})\big\}\Big)\Big)$

The Weisfeiler-Lehman algorithm labels vertices according to their degree and successively refines vertex labels by concatenating them with the sorted labels of their neighbourhood. The algorithm terminates either when a predetermined number of iterations has been reached, or when the two graphs under test have distinct vertex label sets [1]. Unlike the algorithm for regular graphs, the hypergraph algorithm alternately updates vertex and hyperedge labels within each iteration [6]. Efficient implementations of canonical labelling algorithms are provided by tools such as Nauty [13] and Bliss [12]. Given a graph as input, these tools produce a canonical form that can be used to test for isomorphism.

Previously, the Global Benchmark Database identified isomorphism classes using hashes of sorted pairs of literal occurrence counts [7]. This rough approximation has limited precision but has not been rigorously evaluated. We introduce a robust isomorphism-invariant hash based on Weisfeiler-Lehman refinement and evaluate its effectiveness on existing benchmarks.

3 Methodology

We employ one-dimensional label refinement on the bipartite clause–literal incidence graph, inspired by the Weisfeiler-Lehman algorithm for graph isomorphism testing [18]. Algorithm 1 outlines the complete procedure. Each iteration t computes clause labels X_C from literal labels $K_t(l)$ (line 7), aggregates them into accumulators $A[l]$ (line 8), and updates literal labels via complement-aware mixing (line 11). Refinement terminates when variable label counts d stabilize or the iteration cap $t_{\max}$ is reached. All labels X_C, $A[l]$, and $K_t(l)$ are 64-bit unsigned integers in $\mathbb{H} := \{0, \ldots, 2^{64} - 1\}$ with arithmetic modulo 2^{64}. The update rules satisfy two critical invariance properties:

- **Permutation invariance:** Clause and literal labels are aggregated via commutative operations, ensuring the hash remains invariant under reordering of clauses and literals.
- **Polarity invariance:** Literal labels are updated symmetrically for both $K_t(l)$ and $K_t(\bar{l})$. Combined with the canonicalization of variable pairs in the final hash computation, this ensures invariance under polarity flipping.

These invariance properties are essential: they guarantee isomorphic instances produce identical hashes and are maintained throughout execution. Specifically, they govern the computation of permutation-invariant clause labels X_C (line 7, Definition 2), the aggregation of accumulators $A[l]$ (line 8), and the update of literal labels (line 11, Definition 3).

To achieve strong bit dispersion and collision resistance, we employ the mixfast hash function which corresponds to `mix64variant13` by Steele et al. [17].

▶ **Definition 1** (mixfast). *Let $\oplus$ denote bitwise XOR and $\gg$ denote logical right shift. Given a 64-bit word $z_0 \in \mathbb{H}$,* mixfast *is defined as:*

$$z_1 := (z_0 \oplus (z_0 \gg 30)) \cdot \texttt{0xbf58476d1ce4e5b9},$$
$$z_2 := (z_1 \oplus (z_1 \gg 27)) \cdot \texttt{0x94d049bb133111eb},$$
$$\mathsf{mixfast}(z_0) := z_2 \oplus (z_2 \gg 31).$$

The mix function, inspired by SplitMix64 [17], aggregates literal labels within a clause into a permutation-invariant clause label while maintaining strong bit dispersion.

▶ **Definition 2** (mix). *Let $\mathsf{rot}_r(x)$ denote the left rotation of the 64-bit word x by r bits. Let $s_1, s_2 \in \mathbb{H}$ be fixed constants and $r \in \{0, \ldots, 63\}$. For a clause c with literal labels $K_t(l) \in \mathbb{H}$, define*

$$y_l := \mathsf{mixfast}(K_t(l) + s_1), \qquad a := \sum_{l \in c} y_l, \text{ and} \qquad b := \bigoplus_{l \in c} \mathsf{rot}_r(y_l).$$

The clause label is then $\mathsf{mix}(c) := \mathsf{mixfast}\big(a \oplus \mathsf{mixfast}(b + s_2) \oplus |c|\big)$.

The combine function updates literal labels symmetrically for both a literal and its complement, ensuring invariance under polarity flips.

▶ **Definition 3** (combine). *Let $\mathsf{rot}_1(x)$ denote a left rotation by one bit of the 64-bit word x. The function* $\mathsf{combine} : \mathbb{H}^3 \to \mathbb{H}$ *maps an accumulator value $A[l]$, a literal label $K_t[l]$, and the label of its complement $K_t[\bar{l}]$ to*

$$\mathsf{combine}(A[l], K_t[l], K_t[\bar{l}]) = \mathsf{mixfast}\big(\mathsf{mixfast}(A[l]) + K_t[l] + \mathsf{rot}_1(K_t[\bar{l}])\big).$$

For each variable pair $(v, \neg v)$, we canonicalize literal labels as $(\min(a, b), \max(a, b))$ to ensure polarity invariance, sort the canonical pairs lexicographically, and apply `XXH3_64bits` to generate the final hash.

For comparison, we also evaluate two simpler permutation-invariant clause-label alternatives summarized in Table 1. Variant `sort` orders labels within each clause and hashes the result, while `sum` uses simple addition.

4 Evaluation

Our implementation has been integrated into the GBDC extension module [11] of the Global Benchmark Database (GBD) [10] and is publicly available on GitHub [9]. We evaluated our implementation using `scranfilize` [4], confirming invariance under clause and literal

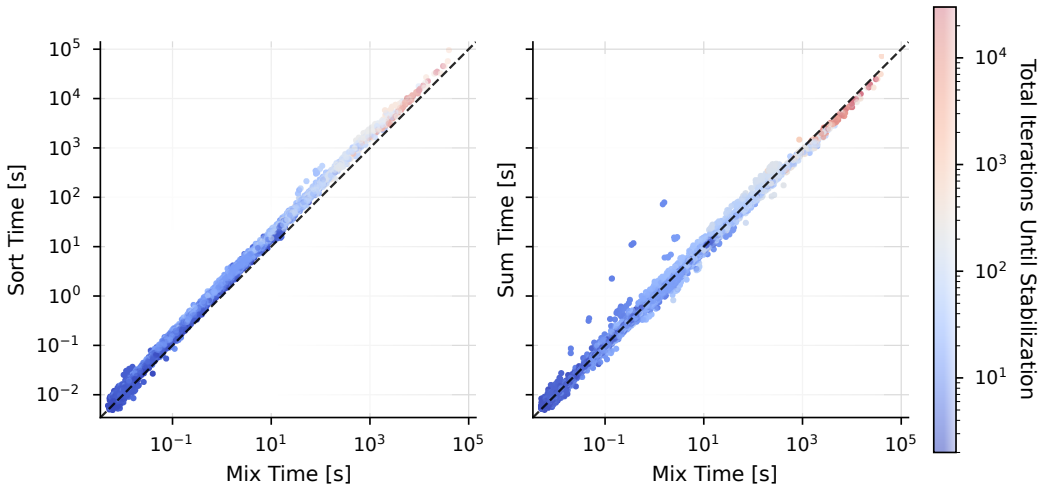

Figure 1 Running times: `Mix` vs. `Sort`/`Sum` (colored by convergence iterations).

reordering, variable renaming, and polarity flips. Randomised scrambling with 20 distinct configurations consistently produced identical hashes, and automated regression tests verify these invariants on every update.

At the time of writing, the GBD contains 31,459 SAT instances from various sources, including those from previous SAT competition tracks. We used this collection to evaluate performance and accuracy across variants and to quantify isomorphism prevalence in SAT competition benchmarks. Measurements were conducted on a machine running Rocky Linux 9.5 (kernel 5.14) with two AMD EPYC 7713 sockets, each with 64 cores and two hardware threads per core, and 2 TiB of RAM, processing 16 SAT instances in parallel.

4.1 Precision–Time Trade-off Across Variants

Our algorithm uses a carefully designed hash function for literal and clause labels that is invariant under permutation and polarity inversion. To motivate this design, we compare the `mix` variant to the two simpler `sort` and `sum` variants.

Table 2 shows the results of executing all three variants until stabilization without imposing a maximum number of iterations. The `mix` and `sort` variants identify the same positive isomorphism checks, while `sum` yields 137 additional false positives. The `mix` variant is approximately 1.7× faster than `sort` on average while matching the performance of the less accurate `sum` variant. These results demonstrate that our method nearly eliminates the precision–speed trade-off: achieving `sort`-level accuracy at close to `sum`-level cost.

Figure 1 further illustrates the running time comparison of the three variants across all benchmark instances. Each point in the scatter plot represents a single instance, with the

Table 1 Clause label construction variants.

Variant	Function	Description
Mix	$X_C := \mathsf{mix}(\{K_t(l) \mid l \in C\})$	Proposed permutation-invariant hash
Sort	$X_C := \mathsf{hash}(\mathrm{sort}(K_t(l) : l \in C))$	Sort labels and hash the resulting list
Sum	$X_C := \sum_{l \in C} K_t(l)$	Simple additive aggregation

Table 2 Clause-labeling variants: positive checks and running times.

Variant	Positives	Average Time (s)	Median Time (s)
Mix	1542	39.48	0.92
Sort	1542	66.08	1.59
Sum	1679	35.51	0.79

Table 3 Positive isomorphism checks and running-time statistics across refinement-iteration limits, where ∞ denotes execution until stabilization.

Max. Iterations	1	2	3	10	20	30	100	∞
Positives	2057	1740	1685	1591	1575	1557	1552	1542
Average Time (s)	2.25	2.72	3.19	5.93	8.77	11.29	19.14	39.48
Median Time (s)	0.26	0.31	0.36	0.62	0.85	0.89	0.92	0.92

x-axis showing the running time of the `mix` variant and the y-axis showing the running time of either the `sort` or `sum` variant. Points are colored according to the number of iterations until stabilization, offering insight into how running time correlates with the algorithm's convergence behavior. The figure again emphasizes that the `mix` variant maintains efficiency comparable to the simpler `sum` variant.

4.2 Convergence and Running Time Analysis

To assess typical refinement effort, we analyze the number of iterations required for stabilization across all instances and the corresponding running times. Table 3 reports positive isomorphism checks for different refinement-iteration limits, together with average and median running times. More iterations improve precision by distinguishing more non-isomorphic instances, at the cost of increased runtime. The final column (∞) corresponds to running without an iteration limit, i.e., until stabilization.

As shown in Figure 2, 50% of instances stabilize by iteration 13 and 90% by iteration 31. The very thin red and orange bands at the top of the cumulative plot indicate that only a small number of instances have particularly long running times; these outliers are typically large instances, where parsing – including the unpacking of hundreds of gigabytes of formula data – dominates total runtime. Based on the 90% threshold, we set a fixed maximum of 31 iterations in the variant integrated into the GBD.

4.3 Isomorphisms in SAT Competition Benchmarks

Table 4 compares non-isomorphic instance counts detected by the previous `IsoHash` and the improved `IsoHash2`, alongside `nominal` and deduplicated (`actual`) instance counts. We list only tracks where the number of non-isomorphic instances under `IsoHash2` differs from the nominal count. Overall, `IsoHash2` yields higher precision than `IsoHash`; for example, in the 2012 Application track it identifies 554 non-isomorphic instances versus 550 for `IsoHash`. The two methods coincide in several tracks, such as 2002 Industrial. The table also highlights nominal-versus-actual discrepancies that have been discussed previously but not yet published; for example, the 2016 Agile track lists 5,000 nominal instances but only 1,580 distinct instances after deduplication.

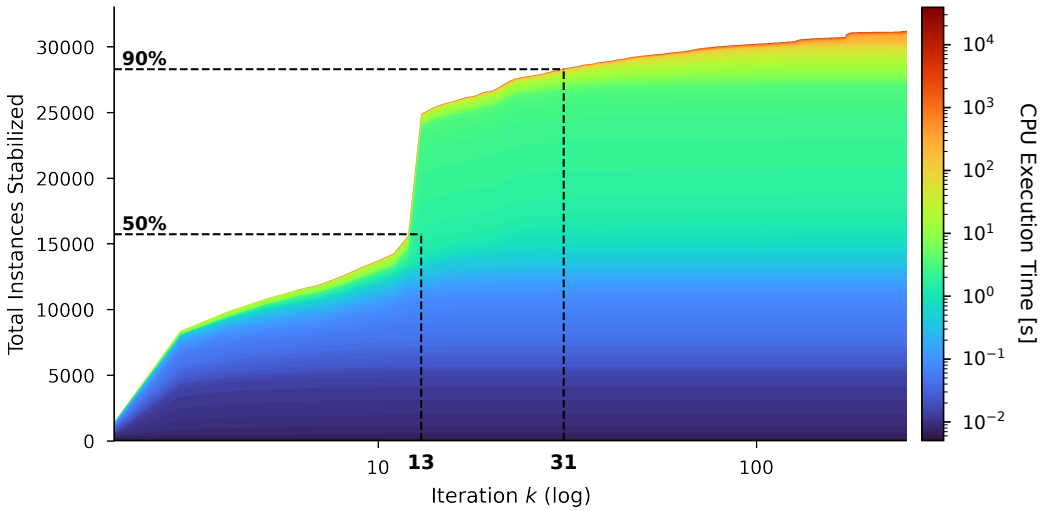

Figure 2 Cumulative number of instances stabilized by iteration k (logarithmic x-axis). Color indicates running time. Dashed lines mark the 50% and 90% stabilization points.

Table 4 Instance Counts and Isomorphism Across SAT Competition Tracks.

Year / Track	Nominal	Actual	IsoHash	IsoHash2
2002 / Handmade	541	541	200	270
2002 / Industrial	189	189	187	187
2003 / Handmade	353	353	314	337
2003 / Industrial	100	100	88	99
2003 / Random	317	317	301	304
2004 / Handmade	228	228	212	225
2004 / Industrial	323	323	102	122
2005 / Crafted	675	675	594	607
2005 / Industrial	212	212	124	135
2007 / Crafted	201	201	191	195
2009 / Crafted	281	281	275	277
2011 / Application	300	300	299	299
2011 / MUS	300	299	293	299
2012 / Application	600	596	550	554
2012 / Crafted	600	600	566	570
2012 / Portfolio	600	599	595	595
2013 / Application	300	300	281	300
2013 / Crafted	300	300	282	297
2014 / Application	300	299	283	299
2014 / Crafted	300	300	258	266
2015 / Main	300	291	282	291
2015 / Parallel	100	96	93	96
2016 / Agile	5000	1580	1579	1579
2016 / Application	300	299	292	298
2017 / Agile	5000	2379	2375	2377
2019 / Main	400	399	357	397
2021 / Crypto	200	200	174	199
2021 / Main	400	400	395	399
2022 / Main	400	400	375	396

5 Discussion

We present a method for efficiently identifying isomorphic SAT instances via Weisfeiler–Lehman refinement. The approach captures CNF structure while maintaining computational efficiency, scaling to large benchmark collections. The `mix` variant nearly eliminates the precision–speed trade-off, delivering `sort`-level accuracy with `sum`-level performance. Empirical results demonstrate that `IsoHash2` substantially reduces false positives compared to degree-sequence hashing, enabling more reliable benchmarking and solver analysis.

Future work includes higher-dimensional Weisfeiler–Lehman refinements to further increase discriminative power. A natural next step is to extend the approach to other combinatorial benchmark collections, particularly MaxSAT variants that preserve objective-function invariances. Finally, for pairs flagged as isomorphic, we aim to produce explicit certificates (e.g., variable and clause mappings) to confirm true positives.

References

1 Markus Anders. *Efficient Algorithms for Symmetry Detection*. PhD thesis, Technische Universität Darmstadt, 2024. doi:10.26083/tuprints-00028257.

2 László Babai. Graph Isomorphism in Quasipolynomial Time [extended abstract]. In *ACM Symposium on Theory of Computing, STOC*, 2016. doi:10.1145/2897518.2897542.

3 László Babai and Ludik Kucera. Canonical Labelling of Graphs in Linear Average Time. In *Symposium on Foundations of Computer Science, SFCS*, 1979. doi:10.1109/SFCS.1979.8.

4 Armin Biere and Marijn Heule. The Effect of Scrambling CNFs. In *Pragmatics of SAT*, 2018.

5 Jin-yi Cai, Martin Fürer, and Neil Immerman. An Optimal Lower Bound on the Number of Variables for Graph Identification. *Combinatorica*, 1992. doi:10.1007/BF01305232.

6 Yifan Feng, Jiashu Han, Shihui Ying, and Yue Gao. Hypergraph Isomorphism Computation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2024. doi:10.1109/TPAMI.2024.3353199.

7 Ashlin Iser. Benchmark Compilation for SAT Competition 2023. In *Proceedings of SAT Competition*, 2023. URL: http://hdl.handle.net/10138/563824.

8 Ashlin Iser and Frederick Gehm. Global Benchmark Database. Software, swhId: swh:1:dir:2553a0109d95559020fb87331f2d49502d4c4c67 (visited on 2026-07-03). URL: https://github.com/udopia/gbdc, doi:10.4230/artifacts.26969.

9 Ashlin Iser and Frederick Gehm. IsoHash2 Implementation in GBDC. https://github.com/Udopia/gbdc/blob/master/src/identify/ISOHash2.h, 2026. Accessed: 2026-03-09.

10 Ashlin Iser and Christoph Jabs. Global Benchmark Database. In *Intl. Conference on Theory and Applications of Satisfiability Testing, SAT*, 2024. doi:10.4230/LIPIcs.SAT.2024.18.

11 Ashlin Iser and Christoph Jabs. Global Benchmark Database (extension module), 2024. doi:10.4230/ARTIFACTS.22457.

12 Tommi Junttila and Petteri Kaski. Engineering an Efficient Canonical Labeling Tool for Large and Sparse Graphs. In *Workshop on Algorithm Engineering and Experiments, ALENEX*, 2007.

13 Brendan D. McKay and Adolfo Piperno. Practical Graph Isomorphism, II. *J. Symb. Comput.*, 2014. doi:10.1016/J.JSC.2013.09.003.

14 Ilya Otpuschennikov, Alexander Semenov, Victor Kondratiev, Daniil Chivilikhin, and Stepan Kochemazov. Benchmarks Encoding Logical Equivalence Checking for Sorting Algorithms. In *Proceedings of SAT Competition*, 2022. URL: http://hdl.handle.net/10138/359079.

15 Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 2011. URL: http://jmlr.org/papers/v12/shervashidze11a.html.

16 Ivor T. A. Spence. sgen1: A Generator of Small but Difficult Satisfiability Benchmarks. *ACM J. Exp. Algorithmics*, 2010. doi:10.1145/1671970.1671972.

17 Guy L. Steele, Doug Lea, and Christine H. Flood. Fast Splittable Pseudorandom Number Generators. In *ACM Intl. Conf. on Object Oriented Programming Systems Languages and Applications, OOPSLA*, 2014. doi:10.1145/2660193.2660195.

18 Boris Weisfeiler and Andrei Leman. Reduction of a Graph to a Canonical Form and an Algebra Arising During this Reduction. *Nauchno-Technicheskaya Informatsiya*, 1968.