Engineering a Preprocessor for Symmetry Detection

Markus Anders
TU Darmstadt, Germany

Pascal Schweitzer
TU Darmstadt, Germany

Julian Stieß
University of Koblenz-Landau, Germany

Abstract

State-of-the-art solvers for symmetry detection in combinatorial objects are becoming increasingly sophisticated software libraries. Most of the solvers were initially designed with inputs from combinatorics in mind (NAUTY, BLISS, TRACES, DEJAVU). They excel at dealing with a complicated core of the input. Others focus on practical instances that exhibit sparsity. They excel at dealing with comparatively easy but extremely large substructures of the input (SAUCY). In practice, these differences manifest in significantly diverging performances on different types of graph classes.

We engineer a preprocessor for symmetry detection. The result is a tool designed to shrink sparse, large substructures of the input graph. On most of the practical instances, the preprocessor improves the overall running time significantly for many of the state-of-the-art solvers. At the same time, our benchmarks show that the additional overhead is negligible.

Overall we obtain single algorithms with competitive performance across all benchmark graphs. As such, the preprocessor bridges the disparity between solvers that focus on combinatorial graphs and large practical graphs. In fact, on most of the practical instances the combined setup significantly outperforms previous state-of-the-art.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms

Keywords and phrases graph isomorphism, automorphism groups, symmetry detection, preprocessors

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.1

Supplementary Material *Software (Source Code)*: `https://github.com/markusa4/sassy`
archived at `swh:1:dir:ba57bf62762f6c5d0bd51ce07862a70df70c8468`

Funding Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EngageS: grant No. 820148).

Acknowledgements We thank Marc E. Pfetsch and Christopher Hojny for giving us further insights into the user-side of symmetry detection software, as well as providing us with the MIP2017 graphs.

1 Introduction

Exploitation of symmetries is an indispensable instrument in a vast number of algorithmic application areas such as SAT [20, 5, 13], SMT [12], QBF [21], CSP [15], ILP [25, 27, 17] and many more. However, in order to exploit symmetries, we have to compute them first.

Many types of objects can be modelled efficiently as graphs, so that the objects' symmetries correspond to the symmetries of the graph. This includes formulas, equation systems, finite relational structures, and many more (see [29]). Hence, computing the symmetries of these objects reduces to computing symmetries of graphs. We refer to the act of computing the symmetries of a graph as *symmetry detection*.

State-of-the-art symmetry detection tools are NAUTY [26], SAUCY [11], BLISS [19], TRACES [26], and DEJAVU [7]. Given as input a vertex-colored graph, they output all the symmetries of the graph. All state-of-the-art tools are based around the so-called individualization-refinement (IR) paradigm. Yet, they substantially differ in the applied search strategies, pruning invariants, symmetry handling, and various other heuristics (see [26, 7]). This is also reflected in diverging performances on different graph classes.

We want to highlight two examples where the diverging performance between the solvers is notable, namely "practical graphs" and "combinatorial graphs". For large practical graphs, such as graphs arising in SAT, QBF, MIP, or road networks, the solver SAUCY outperforms all other solvers significantly (see, e.g., the results in [5] or the benchmarks of this paper in Section 9). Indeed, designed with satisfiability-checking in mind, SAUCY has been delicately engineered specifically for these types of graphs. Intuitively, graphs arising from practical applications tend to be large in size but comparatively simple in their structure. On the other hand, on almost all graph classes that are difficult relative to their size (e.g., projective planes, CFI graphs, and other regular combinatorial objects) TRACES and DEJAVU readily outperform other solvers due to their more sophisticated search strategies (see [7] and [26] for a more nuanced discussion). In Figure 1, we demonstrate the large disparity between SAUCY and DEJAVU on a difficult graph class from combinatorics and a class of practical graphs.

Only having solvers available that are geared towards specific types of graphs is of course an undesirable situation. This for example means we have to choose a solver and thus understand the type of input we are faced with. Also, we will struggle with inputs that are combinations of the different kinds of graphs. Quite naturally, it is desirable instead to have a single solver performing well on all graphs.

A commonly used paradigm to make solvers for computational problems more widely applicable is to add a preprocessor. The use of preprocessors has indeed already led to countless success stories, in particular in SAT, QBF or MaxSAT [14, 10, 24]. In these applications, it is nowadays standard to apply a preprocessor to all inputs. In contrast to this, to date, no preprocessor has been available for symmetry detection. In fact McKay and Piperno [26] explicitly highlight that in their opinion "graphs of [particular types] ought to be handled by preprocessing" before using their tools.

Beyond increasing performance, there are various other benefits to having a preprocessor. Firstly, the problem of initially simplifying the graph can be tackled independently from the design of the main solver. This is especially desirable since implementations of state-of-the-art symmetry detection solvers are complex and detailed descriptions of the inner workings largely unavailable. Secondly, in turn, a preprocessor could even reduce the complexity of solver implementations if certain cases are reliably handled before running the solver. Lastly, implementing strategies in a common preprocessor makes them available to all the solvers simultaneously.

Given the lack of an existing preprocessor for symmetry detection, TRACES, for example, has complicated subroutines that simplify some low-degree vertices before (and sometimes during) the computation (see the implementation [2]). Overall, the question is whether it is possible to design a common preprocessor that can simplify inputs and is beneficial to all state-of-the-art solvers.

Contribution. We implement the first preprocessor SASSY for symmetry detection. It is compatible by design with all state-of-the-art symmetry detection tools. Our benchmarks (Section 9) corroborate that solver configurations using the preprocessor significantly outperform state-of-the-art on many practical graph classes. At the same time, the preprocessor introduces only a negligible overhead.

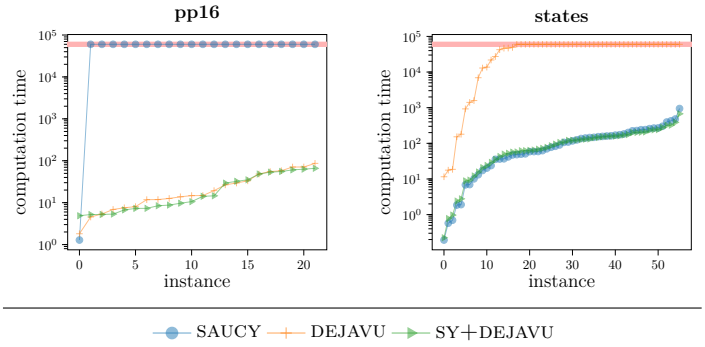


Figure 1 Comparing solvers on difficult combinatorial graphs (**pp16**) and large practical graphs (**states**). Timeout is 60*s* (red bar). SY+DEJAVU refers to DEJAVU with the preprocessor of this paper.

The preprocessor bridges the disparity that exists between solvers that focus on difficult combinatorial graphs (TRACES, DEJAVU, BLISS, NAUTY) and those that focus on large practical graphs (SAUCY). Through the use of the preprocessor, the former kind of solvers now outperform SAUCY on most practical graphs.

Techniques. The preprocessor implements mainly techniques to handle graphs that are sparse, both in the input and output (e.g., practical graphs). In particular, it is made up of the following building blocks which we discuss throughout the paper:

1. A framework to capture reduction techniques for input graphs. In particular, it captures the reconstruction of symmetries from the reduced graph back to the input graph, both theoretically and practically (Section 4).
2. A technique to efficiently remove vertices of degree 0 and 1 (Section 5.1 and Section 5.2).
3. Partial removal of degree 2 vertices avoiding the introduction of colored or directed edges (Section 5.3).
4. An individualization-refinement-based probing technique for "sparse automorphisms" (Section 6).
5. Exploiting connected components and homogeneous connections using the concept of quotient graphs (Section 7).

While SASSY is the first universal preprocessor for symmetry detection, we want to remark that a flavor of (2) is already implemented in TRACES. All the techniques other than (2) are novel contributions, however, we do want to mention that (4) and (5) draw some inspiration from existing techniques of solvers. We explain this in detail in the respective sections.

2 Philosophy of the Preprocessor

When designing a preprocessor, one of the main challenges is to map out which techniques and methods fall within the responsibility of the preprocessor and which task should be resolved by the main algorithm. Another delicate matter are the preprocessor/main solver and the user/preprocessor interfaces. In the design of our preprocessor we were guided by conceptual principles as well as technical requirements.

Conceptual principles. On a conceptual level, our goal is to design efficient preprocessing subroutines that simplify the task of computing symmetries. Naturally, a preprocessor should only apply procedures that are comparatively fast in relation to the running time of the main algorithm.

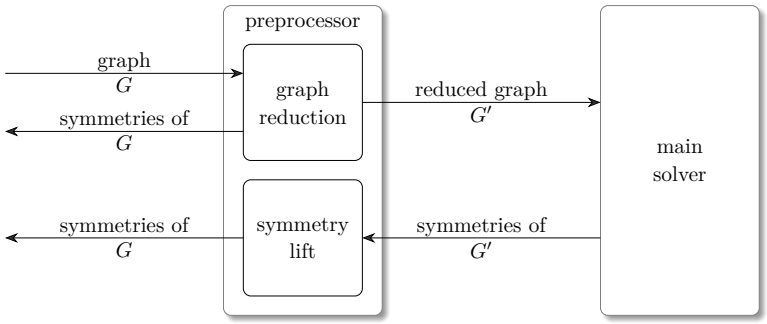


Figure 2 Our proposed preprocessor/main solver and user/preprocessor interfaces. The preprocessor may already determine some (or all) symmetries of G during graph reduction. The reduced instance is then passed on to the main solver.

The design of our preprocessor is centered around the so-called *color refinement* algorithm. Color refinement is a powerful heuristic for symmetry detection. It is continuously and repeatedly applied in all state-of-the-art solvers. Thus, procedures that run within or close to color-refinement-time are safe to apply.

The general idea of the preprocessor is to remove substructures of the graph that are already "basically resolved" by an application of color refinement. The main difficulty lies in detecting and exploiting these substructures as efficiently as possible. Essentially, any part that can be handled efficiently ought to be carefully handled using precisely the right technique.

Overall, we need to balance efficiency, effectiveness, and generality for our subroutines.

Technical requirements. On a technical level, we want our preprocessor to be compatible with all state-of-the-art solvers. Hence, we need to use an interface that is universal for all the existing tools. All tools read vertex-colored graphs and output symmetries. Hence, this is the interface that the preprocessor uses as well.

The preprocessor reads a vertex-colored graph and outputs a reduced vertex-colored graph passed to a main solver. Moreover, the preprocessor may already determine some or all of the symmetries, and immediately outputs these to the user. There is one more technicality: symmetries of the reduced graph which are computed by the main solver are, by definition, not symmetries of the original graph. To rectify this, the preprocessor employs a backward-translation (i.e., a form of postprocessing) to lift symmetries that were discovered by the main solver back to being symmetries of the original input graph.

Our design is illustrated in Figure 2.

3 Preliminaries

A graph G is a finite, simple, undirected graph, unless stated otherwise. The neighborhood of a vertex v is denoted $N(v)$, its *degree* is $\deg(v) := |N(v)|$. For a set of vertices $V' \subseteq V(G)$ the *neighborhood* is the set $N[V'] := (\bigcup_{v \in V'} N(v)) \setminus V'$.

A *coloring* of a graph G is a map $\pi\colon V(G) \to \mathcal{C}$ from the vertices to some set of colors. A (color) *class* C is a set $\pi^{-1}(c)$ of vertices of the same color. A coloring π is referred to as *discrete* whenever π is injective. In other words, in a discrete coloring each vertex has its own unique color. Unless stated otherwise, we work with *colored graphs* $G = (V, E, \pi)$ which consist of vertex set V, edge set E, and a coloring π. Slightly abusing notation the

pair (G, π) for an uncolored graph $G = (V, E)$ is identified with (V, E, π). For a subset of the vertices $V' \subseteq V$ the *induced subgraph* of $G = (V, E, \pi)$ is $G[V'] = (V', E', \pi|_{V'})$ where $E' = \{e \in E \mid e \subseteq V' \times V'\}$.

A bijection $\varphi : V \mapsto V$ is called an *automorphism* (symmetry) whenever $(\varphi(V), \varphi(E)) = (V, E)$ (applying φ element-wise to the vertices in the edges of E). If G is colored, φ also has to respect colors (i.e., satisfy $\pi(\varphi(v)) = \pi(v)$). The number of automorphisms can be exponential in the size of the graph. The symmetries form a permutation group under the composition operation. The *automorphism group* containing all automorphisms of a (colored) graph G is $\mathrm{Aut}(G)$. The *support* of an automorphism $\varphi \in \mathrm{Aut}(G)$ is $\mathrm{supp}(\varphi) := \{\varphi(x) \neq x \mid x \in V(G)\}$, i.e., vertices not fixed by the automorphism. A subset of automorphisms $S \subseteq \mathrm{Aut}(G)$ is a *generating set* of $\mathrm{Aut}(G)$, whenever exhaustively composing permutations of S leads to all elements of $\mathrm{Aut}(G)$. We write $\langle S \rangle = \mathrm{Aut}(G)$. This enables a concise encoding of $\mathrm{Aut}(G)$. Solvers generally only output a generating set of $\mathrm{Aut}(G)$.

3.1 Color Refinement

The color refinement algorithm is a well-studied procedure [8, 9, 26]. For a colored graph it splits apart colors in a specific way to produce a "finer" coloring. Crucially, this process does *not* change the symmetries of the graph.

Formally, a coloring π of a graph is *equitable* if for all pairs of (not necessarily distinct) color classes C_1, C_2, all vertices in C_1 have the same number of neighbors in C_2 (i.e., $|N(v) \cap C_2| = |N(v') \cap C_2|$ for all $v, v' \in C_1$.) Given a coloring π, color refinement computes an equitable refinement π' (i.e., an equitable coloring π' for which $\pi'(v) = \pi'(v')$ implies $\pi(v) = \pi(v')$). In fact, it computes the coarsest equitable refinement. Crucially, automorphisms of $G = (V, E, \pi)$ are also automorphisms of $G = (V, E, \pi')$ (and vice versa). It is thus beneficial and routine to work with π' instead of π. Color refinement can be implemented in such a way that it admits a worst case running time of $\Theta((n + m)(\log n))$ (see [9]). From an implementation perspective it is the most crucial subroutine and therefore highly engineered.

3.2 Quotient Graph

For an equitable coloring π of an (otherwise uncolored) graph G, the *quotient graph* $Q(G, \pi)$ captures information regarding the number of neighbors that vertices in one color class have in another color class. A quotient graph is a complete directed graph in which every vertex has a self-loop. The vertex set of $Q(G, \pi)$ is $V(Q(G, \pi)) := \pi(V(G))$, i.e., the set of colors of vertices under π. The vertices of $Q(G, \pi)$ are colored with the color they represent in G. We color the edge (c_1, c_2) with the number of neighbors a vertex color c_1 has of color c_2 (possibly $c_1 = c_2$). Recall that, since π is equitable, all vertices of c_1 have the same number of neighbors in c_2. Two graphs are *indistinguishable by color refinement* if and only if their quotient graphs with respect to the coarsest equitable coloring are equal (see e.g. [8]).

4 A Toolbox for Reducing Graphs

We now embark on our journey of describing techniques that simplify a graph for symmetry detection. The goal is always to efficiently reduce the number of vertices and edges of the graph. However, whenever we alter the graph, we need to make sure that either no symmetries are lost, or that we output the symmetries that would be lost immediately. Furthermore, we have to ensure that after preprocessing is done symmetries of the reduced graph can be

mapped back to symmetries of the original graph. After all, we are interested in symmetries of the original graph. In order to ease this process, we first lay out some general techniques that we use throughout the paper.

The first type of technique we describe modifies an input graph G on vertex set V to another graph G' with vertex set $V' \subseteq V$ so that

1. $\mathrm{Aut}(G)|_{V'} \subseteq \mathrm{Aut}(G')$ (symmetry preservation) and
2. $\mathrm{Aut}(G)|_{V'} \supseteq \mathrm{Aut}(G')$ (symmetry lifting) hold.

Here by $\mathrm{Aut}(G)|_{V'}$ we mean the set of maps obtained by restricting the domain of each $\varphi \in \mathrm{Aut}(G)$ to V' (and the range to $\varphi(V')$). If conditions (1) and (2) hold, V' must also be invariant under $\mathrm{Aut}(G)$.

Under these conditions the restriction to V' is a natural homomorphism $p\colon \mathrm{Aut}(G) \to \mathrm{Aut}(G')$. The orbit-stabilizer theorem (see [18, Theorem 2.16]) implies then that if $S' \subseteq \mathrm{Aut}(G)$ is a set of lifts of a generating set S of $\mathrm{Aut}(G')$, i.e. $p(S') = S$, then $\mathrm{Aut}(G) = \langle S', \ker(p)\rangle$ (where $\langle\Gamma\rangle$ denotes the group *generated* by Γ, see [30]). Here $\ker(p) = \{\varphi \in \mathrm{Aut}(G) \mid p(\varphi) \neq 1\}$ is the *kernel* of p and 1 denotes the identity.

Overall this enables us to separate the computation of $\mathrm{Aut}(G)$ into computing automorphisms of the removed parts of the graph and the automorphisms of the reduced graph. Crucial for the techniques is now that G' and a generating set of $\ker(p)$ can be efficiently computed from G, and that the set of lifts S' can be efficiently computed from a generating set of $\mathrm{Aut}(G')$. In particular, we require an efficient postprocessing technique for lifting of automorphisms to parts that were reduced, which is described in the following.

Canonical Representation Strings. During preprocessing, the parts we remove from the original graph might be symmetrical to (i.e., in the same orbit as) other parts of the graph. So, after symmetries of the reduced graph have been computed, we need to lift symmetries of the reduced graph to symmetries of the original graph. In particular, the lifted symmetries must map all the removed parts correctly. To simplify the lifting of symmetries we introduce *representation strings* associated with the remaining vertices. These encode the nature (i.e., the "isomorphism type") of the vertices that were removed. The encoding is stored in the color of a suitable vertex that remains. If a remaining vertex is then mapped to another vertex, the corresponding subgraphs represented by the strings are then mapped to each other in a canonical way.

We define this process formally through a *representation mapping* $\mathcal{R}(v) : V \mapsto V^*$ from the vertices to sequences of vertices as follows. Assume we have a graph $G := (V, E, \pi)$ which is reduced to $G' := (V', E', \pi')$ with $V' \subseteq V$ and $E' \subseteq E$. We require the following:

1. It holds that $\mathcal{R}(v) := vS$ with $S \in V^*$ for all $v \in V'$, i.e., each remaining vertex must represent itself first.
2. It holds that $\mathcal{R}(v) := \epsilon$ for all $v \in V \setminus V'$, i.e., a removed vertex does not represent any vertex.
3. For each deleted vertex $v \in V \setminus V'$ there is at most one $v' \in V'$ and at most one $i \in \mathbb{N}$ such that $v := \mathcal{R}(v')_i$, i.e., each deleted vertex is represented by at most one remaining vertex, once.

For each automorphism of the remaining graph $\varphi \in \mathrm{Aut}(G')$ we now define its *lifted bijection* $\varphi_{\mathcal{R}}(v) \in \mathrm{Sym}(V)$ (the symmetric group on V). First, we require that $\varphi(v) = v' \implies |\mathcal{R}(v)| = |\mathcal{R}(v')|$ holds, otherwise we can not construct a lifted bijection. We define $\varphi_{\mathcal{R}}(v) :=$

$$\begin{cases} \varphi(v) & \text{if } v \in V' \\ \mathcal{R}(\varphi(v'))_i & \text{if } v = \mathcal{R}(v')_i \text{ for } v' \in V', i \in \mathbb{N} \\ v & \text{if } v \neq \mathcal{R}(v')_i \text{ for all } v' \in V', i \in \mathbb{N}. \end{cases}$$

We call $\mathcal{R}$ a *canonical representation mapping* if $\varphi_{\mathcal{R}} \in \mathrm{Aut}(G)$ for all $\varphi \in \mathrm{Aut}(G')$.

We note by definition, canonical representation mappings can be chained, i.e., if we reduce a graph G multiple times, we can simply apply the respective canonical representation mappings in reverse until we reach an automorphism of G. We can even rewrite chained canonical representation mappings into a single map by essentially composing the functions. (More accurately we have to interpret strings of strings as simple strings using concatenation.)

Sparse Automorphisms and Restoration. A concept that we implicitly use throughout the following sections is sparse encodings of automorphisms. A conventional way to do this is the cycle notation of permutations, i.e., store only for each non-fixed element its image [18]. The precise encoding used is of no importance, however. Crucially, automorphisms ought to be encoded using space that is proportional to the size of their support, i.e., in $\mathcal{O}(|\operatorname{supp}(\varphi)|)$.

Using a canonical representation mapping $\mathcal{R}$ and sparse automorphism encodings, automorphisms of a reduced graph G' can be efficiently lifted to automorphisms of the original graph G. Indeed, lifts can be computed in time (and in space) linear in the size of the support of the lift, by replacing vertices by their represented strings.

▶ **Fact 1.** *Given $\varphi \in \operatorname{Aut}(G')$, the lift $\varphi_{\mathcal{R}} \in \operatorname{Aut}(G)$ can be computed in time $\mathcal{O}(|\operatorname{supp}(\varphi_{\mathcal{R}})|)$.*

Let us remark that often canonical representations in fact ensure that lifted supports are as small as possible. We say that a representation mapping $\mathcal{R}$ *respects kernel orbits* if it has the property that $v_1 \in \mathcal{R}(v) \Leftrightarrow v_2 \in \mathcal{R}(v)$ whenever v_1 and v_2 are in the same orbit of $\ker(p)$. All representations we describe subsequently respect kernel orbits.

▶ **Fact 2.** *If $\mathcal{R}$ respects kernel orbits then $p(\psi) = \varphi$ implies that $|\operatorname{supp}(\varphi_{\mathcal{R}})| \leq |\operatorname{supp}(\psi)|$.*

We should remark that none of the state-of-the-art solvers except for SAUCY feature an interface for sparse automorphisms, i.e., an interface that enables access to an automorphism in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Instead, access is only possible in $\Omega(|V|)$. If a user-application uses the interface for sparse automorphisms correctly, this can yield substantial running time benefits on graphs that contain a large number of sparse automorphisms (which is the case for many practical graphs). Most solvers internally incur a cost of $\Omega(|V|)$ to handle automorphisms anyway, in turn making the sparse interface unnecessary. Since this is not true for our preprocessor and to ensure potential running time benefits to user-applications, automorphisms found by the preprocessor are of course accessible in a sparse manner.

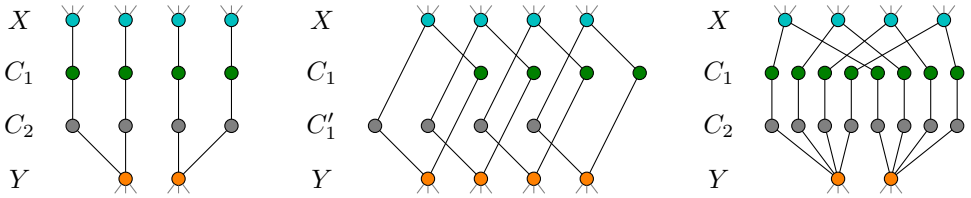
5 Removing low degree vertices

The first class of efficient reduction techniques we describe removes vertices of low degree. We propose strategies for vertices of degree $0, 1$ and 2. Techniques for preprocessing vertices of degree 0 and 1 can also be found in the implementation of TRACES [2]. The TRACES implementation for degree $0, 1$ differs from our proposed strategy in that it does *not* compute color refinement before removing degree 0 and 1 vertices.

5.1 Degree 0 Vertices

Preprocessing vertices of degree 0 (and analogously $n - 1$) is simple. The algorithm detects color classes consisting of vertices of degree 0. We let V' be the set of vertices of degree larger than 0. By simply removing vertices of degree 0 and not representing them in $\mathcal{R}$ at all, $\mathcal{R}$ indeed defines a canonical representation mapping.

The kernel $\ker(p)$ of the restriction p onto V' is computed as follows. For each color class of degree 0 vertices in G we output generators for the symmetric group on the class.



(a) Unique endpoints. **(b)** Obfuscated matching. **(c)** Obfuscated edge flip.

Figure 3 Reducible degree 2 patterns.

5.2 Degree 1 Vertices

Exhaustively removing all vertices of degree 1 (and analogously $n-2$) essentially removes all tree-like appendages from graphs. It is well-known that applying color refinement produces the orbit partitioning on these tree-like appendages – with the notable exception of not determining whether the roots of these appendages are in the same orbit or not.

We can remove degree 1 vertices recursively. Let G be a graph that contains degree 1 vertices. We describe G' and $\mathcal{R}$ where we remove a color class of degree 1 vertices.

Let C denote such a color class of degree 1 vertices. Since the coloring is equitable, all neighbors of vertices of C are in the same color class P. In case $P = C$ we have connected components of size 2. This case can be handled similar to the reduction of degree 0 vertices, so we assume $P \neq C$. We partition C into classes $C_1, \dots, C_m$ where $c \in C_i$ is adjacent to $p_i \in P$. For the representation mapping, we set $\mathcal{R}(p_i) := p_i C_i$ (where C_i may appear in arbitrary order). We set $G' := G \setminus \{C\}$. The coloring π remains unchanged. Note that π is still an equitable coloring for G'. The kernel $\ker(p)$ is the direct product of the symmetric group $\mathrm{Sym}(C_i)$ for each $i \in \{1, \dots, m\}$ (and points outside C are fixed). The process can then be repeated until all vertices of degree 1 are removed.

By construction, the reduction is symmetry preserving and symmetry lifting, thus it holds that $\mathrm{Aut}(G) = \langle S', \ker(p) \rangle$. As before, S is a generating set for $\mathrm{Aut}(G')$ and S' a corresponding set of lifts.

5.3 Degree 2 Vertices

If we were to allow graphs produced by our preprocessor to contain directed, colored edges, there is a simple reduction that removes all vertices of degree 2: we may encode the multiset of paths between two vertices v_1 and v_2 with $\deg(v_i) \geq 3$ whose internal vertices all have degree 2 as one directed, colored edge between v_1 and v_2 (see also [22, Proof of Lemma 15]).

There are, however, drawbacks to this approach: most solvers do not implement directed and colored edges. Since we want our preprocessor to be compatible with all modern solvers, this immediately disallows the use of directed, colored edges. Even when they do, using directed and colored edges comes at the price of additional overhead [28]. Intuitively, while removing all degree 2 vertices can cause a significant size-reduction, some of the complexity of the removed path is only shifted into the color encoding of the edges. In turn, we require refinements to take into account edge colors. This complicates color refinement, the central subroutine.

For these reasons, if possible, we prefer to remove degree 2 vertices in a way that does not require the introduction of directed or colored edges.

Non-branching paths with unique endpoint. We describe a heuristic which we found to be often applicable in practical data sets. It encodes paths with internal vertices of degree 2 that run between two color classes by a set of edges connecting the endpoints directly. However, it only does so if the set of paths can be reconstructed unambiguously from the set of edges. In particular, the inserted edges may not interfere with existing edges.

We detect paths of length t between distinct color classes X and Y whose internal vertices have degree 2. In each vertex of X exactly one such path should start (see Figure 3a). More formally, suppose $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ are colors so that (1) vertices in X do not have neighbors in Y, (2) for $i \in \{1, \ldots, t\}$ vertices in C_i have degree 2, (3) for $i \in \{1, \ldots, t\}$ vertices in C_i have a neighbor in C_{i-1} and C_{i+1}, and (4) every node in X has exactly one neighbor in C_1. Then we define $G' = (V', E')$ via $V' \coloneqq V - (C_1 \cup \cdots \cup C_t)$ and $E' \coloneqq E(G[V']) \cup E''$, where E'' consists of pairs (x, y) for which there is a path $(x, c_1, \ldots, c_t, y)$ with $c_i \in C_i$. The corresponding representation map is $\mathcal{R}(x) = x c_1 c_2 \cdots c_t$, where $(x, c_1, \ldots, c_t, y)$ is the unique path from x to some vertex $y \in Y$ with $c_i \in C_i$.

Note that the newly introduced edges E'' form a biregular bipartite graph between X and Y in which vertices of X have degree 1. It is not difficult to check that this yields a canonical representation map that respects kernel orbits.

Obfuscated Matchings. The preprocessor has special fast code for the particular case in which $|X| = |Y|$. In this case E'' encodes a perfect matching between X and Y.

A slight extension of the technique checks for other choices of C_i whether they also satisfy the required properties and yield exactly the same matching E''. In fact, if there is another matching via color classes $C'_1, \ldots, C'_{t'}$ between X and Y which encodes E'' , we also delete vertices in the C'_i (see Figure 3b). The special purpose code uses arrays and can efficiently check whether matchings coincide.

We should mention that in the implementation, we only perform the check for paths of length $t = 1$ for obfuscated matchings. It turns out that the special case of $t = 1$ and in fact multiple such paths encoding the same matching is very common in particular on the MIP and SAT benchmarks.

Obfuscated Edge Flip. A case that also can be handled efficiently and is not covered by previous techniques is where X and Y are connected by $|X||Y|$ equally-colored, unique paths. In this case, each vertex $x \in X$ is connected to all $y \in Y$ by a path (see Figure 3c). It is easy to see that deleting all such paths is both symmetry preserving and symmetry lifting (this is related to the edge flip described in Section 7.1).

Formally, suppose $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ are colors so that (1) for $i \in \{1, \ldots, t\}$ vertices in C_i have degree 2, (2) for $i \in \{1, \ldots, t\}$ vertices in C_i have a neighbor in C_{i-1} and C_{i+1}, and (3) every node in X has exactly $|Y|$ neighbors in C_1, where the corresponding paths end in all $y \in Y$. The technique in turn removes all $C_0, C_1, \ldots, C_t$ from the graph.

Let us now consider computing the lift of this reduction. Unfortunately, canonical representation strings are not sufficient to express the lift: we need to determine how $C_0, C_1, \ldots, C_t$ are mapped, and this depends on *both* the vertices of X and Y. We can not simply attach $C_0, C_1, \ldots, C_t$ to the canonical representation strings of one of the color classes. However, if we know how both X and Y are mapped, it is trivial to reconstruct the original symmetry: assume a symmetry maps $x \in X$ to x' and $y \in Y$ to y'. This just means that in the lift, we need to map the path connecting x to y to the path connecting x' to y'. Hence, the lift can still be computed very easily and efficiently.

In the implementation, we do write vertices of $C_0, C_1, \ldots, C_t$ into both the representation strings of X and Y, breaking the formal requirement of not having double entries. We use an encoding trick to denote the double entries, which triggers special code during the reconstruction of the symmetries.

Again, these types of degree 2 vertices can often be found in graphs stemming from SAT.

6 Probing for Sparse Automorphisms

We propose a strategy for probing for sparse automorphisms. If successful and automorphisms are discovered, we "divide them out", breaking the symmetry by *individualization*, i.e., giving a vertex a unique color. We give a brief, high-level description. The full description can be found in Appendix A.

Our strategy is inspired by a heuristic of SAUCY: for two colorings π_1, π_2 we may check whether interchanging vertices in color classes of size 1 (i.e., *singleton vertices*) of corresponding singleton colors and fixing all other vertices yields an automorphism of the graph. More formally, we define the permutation $\varphi_{\pi_1,\pi_2}(v) :=$

$$\begin{cases} v & \text{if } |\pi_1^{-1}(\pi_1(v))| \neq 1 \vee |\pi_2^{-1}(\pi_2(v))| \neq 1 \\ \pi_2^{-1}(\pi_1(v)) & \text{otherwise.} \end{cases}$$

Then, we may simply check whether φ_{π_1,π_2} is indeed an automorphism of G. Indeed, this check can be computed in time $\mathcal{O}(\Sigma_{v \in \text{supp}(\varphi_{\pi_1,\pi_2})} 1 + \deg(v))$.

SAUCY performs the check for local automorphisms during its depth-first search of its backtracking tree. It can then store the information about the automorphism and internally exploit its existence. For preprocessing purposes, however, we want to make the graph simpler or smaller.

Our probing strategy chooses a color class C of the graph and then concurrently performs two arbitrary root-to-leaf walks on the backtracking tree (*individualization-refinement* tree) that is also used by main solvers. Through the design of the backtracking procedure, each walk has a natural corresponding coloring (e.g., π_1 and π_2). We then continuously check whether the two walks already imply an automorphism (using φ_{π_1,π_2}). If, using this strategy, we find enough automorphisms to determine that C is equivalent to an orbit, we can individualize a vertex of C, thus simplifying the graph.

7 Exploiting the Quotient Graph

We now introduce another set of techniques which make use of the quotient graph $Q(G, \pi)$.

7.1 Edge Flip and Removal of Trivial Components

First, we describe how to efficiently *flip edges* between color classes. Let C_1, C_2 be two distinct color classes of π. Assume they are connected by m edges. The maximum number of edges between C_1 and C_2 is $|C_1||C_2|$. If $m > {}^{|C_1||C_2|}/_2$, we can flip every edge to a non-edge, and every non-edge to an edge, reducing the total number of edges in the graph. Since this operation is isomorphism-invariant and reversible, the automorphism group of the graph does not change.

When applying edge flips repeatedly and exhaustively, singleton vertices become vertices of degree 0. In fact, instead of performing edge flips in which singletons are involved, we can remove singletons directly without changing the automorphism group.

We want to remark that in the implementation, we use one canonical representation mapping to keep track of all removed vertices. This also includes removed singletons. Hence, we use string representations throughout all the techniques described in the paper. In addition to acting as a global canonical representation mapping, we also allow a renaming of vertices, which enables us to map all remaining vertices into the interval $\{1, 2, \ldots, n\}$, whenever n vertices remain.

7.2 Connected Components

A strategy more general than removing singletons is to exploit connected components of the quotient graph.

Consider the quotient graph $Q = Q(G, \pi)$ of a graph G with respect to a vertex coloring π. The (weakly) connected components of Q partition the vertex set of G into parts that are homogeneously connected. This allows us to treat components independently:

▶ **Lemma 1.** *If $D_1, \ldots, D_t$ are the connected components of the quotient graph $Q(G, \pi)$ then $\mathrm{Aut}(G, \pi) = \Pi_{i=1}^{t} \mathrm{Aut}((G, \pi)[D_i])$.*

By flipping edges between two color classes we can only ever shrink the components of $Q(G, \pi)$. It is therefore beneficial to first exhaustively flip edges and then consider connected components (see also [23]).

These types of components have previously been employed for isomorphism and automorphism testing [16, 19]. (In these contexts flips are not employed but rather edges in the quotient graph are characterized by non-homogeneous connections, which is equivalent.)

Regarding the implementation, we compute the connected components of the quotient graph without explicitly computing the quotient graph. We first perform edge flips for all fully connected color classes, i.e., whenever the number of edges between C_1, C_2 equals $|C_1||C_2|$. Then, we modify a basic algorithm for computing connected components as follows: usually, the algorithm determines for a vertex v its neighborhood $N(v)$ and adds this neighborhood to the connected component of v. Our modification simply also adds $\pi^{-1}\pi(v)$ in addition to $N(v)$ (i.e., it adds entire color classes). In turn, the algorithm gives us a partition of the vertices into the components of the quotient graph.

We use this to perform the probing strategy of Section 6 for each component of the quotient graph separately. We want to mention that after preprocessing is done, we could, theoretically, also use the components of the quotient graph to make independent calls to the main solver on the subgraphs induced by the components. These would, in turn, be smaller, and their handling could be parallelized. However, in our testing, after preprocessing is done, usually only one component is left, or there is one very large component and several smaller ones. We thus, at least so far, did not find it beneficial to use independent solver calls.

8 Scheduling of Techniques

We now describe when and how the preprocessor combines the techniques described in the previous sections.

The first step of the preprocessor is to apply color refinement to produce an equitable coloring. The coloring remains equitable throughout the entire algorithm, by reapplying color refinement whenever necessary (i.e., for the probing techniques). We also continuously remove singletons. Beyond this, our implementation allows the user to freely specify a schedule for the various techniques.

The schedule used to produce the benchmarks is as follows. We remove vertices of degree 0 and 1, and apply the heuristics described for vertices of degree 2. Next, we flip edges and apply probing for sparse automorphisms while making use of quotient graph components. Lastly, we repeat the schedule as long as the graph still contains vertices of degree 0 or 1 and the number of vertices of the graph shrunk by at least 25%. Note that this ensures that the schedule is only repeated at most a logarithmic number of times in the original graph size.

The implementation is called SASSY. It is implemented in C++ and uses the color refinement of DEJAVU (which is itself an amalgam of color refinement implementations in TRACES and SAUCY). The implementation is open source and freely available at [3].

9 Benchmarks

We split the benchmark section into three parts: first, we check whether applying the preprocessor speeds up state-of-the-art solvers on graph classes where the preprocessing techniques are supposedly effective. At the same time we check whether we introduced excessive overhead on graphs where the techniques are not effective. Secondly, we compare the performance of solver configurations using the preprocessor to state-of-the-art SAUCY and TRACES on a wide range of practical data sets. Thirdly, we analyze the separate impact of each of the different techniques used in the preprocessor (see Appendix D).

Whenever we apply the preprocessor followed by an execution of a main SOLVER, we write SY+SOLVER. The reported running time for a configuration SY+SOLVER is always the time used for preprocessing *and* solving the graph. All benchmarks were run on a machine featuring an Intel Core i7 9700K, 64GB of RAM on Ubuntu 20.04. We used NAUTY/TRACES 2.6, SAUCY 3.0, BLISS 0.73 and DEJAVU 1.2 (1% error bound and 4 threads for DEJAVU, all other tools are only able to run single-threaded). We ran all benchmarks 3 consecutive times in order to check whether running times are stable. We report the average and standard deviation.

Conventionally, the way to test symmetry detection solvers is to first randomly permute all given benchmark graphs [26, 7]. However, we feel that for many of the practical graphs, it is not clear whether this is the right way to test the tools: the initial order is often not arbitrary and may indeed encode information. For example in SAT, usually "literal" vertices and "clause" vertices are never mixed but appear as contiguous blocks of vertices. While this does not immediately help the symmetry detection process, aspects such as cache-efficiency might be affected. Therefore, we ran all benchmarks both ways: in the conventional manner of randomly permuting the instances (denoted with **(p)**), as well as using unaltered instances. Benchmarks for permuted graphs are in this section, while results for non-permuted graphs are in Appendix B. Overall, the results for both agree.

9.1 Preprocessed versus Unprocessed

We prepared two collections of graphs to test the impact of applying the preprocessor for each solver. **pract** contains practical graphs with a lot of exploitable structure for the preprocessor. On the other hand, the set **comb** contains combinatorial graphs where there is no or very little exploitable structure. In the following, we describe how we composed both sets.

Set "pract". The goal of this set is to measure whether preprocessing is worthwhile for a given solver on graphs where there is a lot of exploitable structure. Thus, this set contains practical graphs. Note that we test practical graphs much more thoroughly in the next

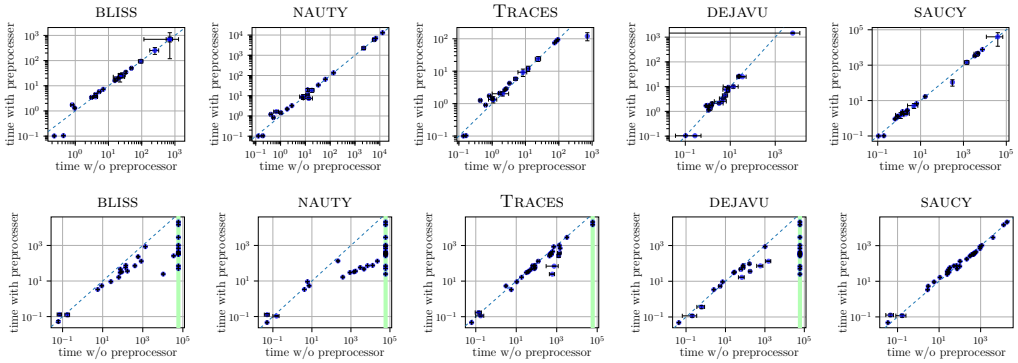


Figure 4 Solvers with SASSY vs. solvers without SASSY on **comb (p)** (top) and **pract (p)** (bottom). Timeout is 60*s*. The green bar shows instances that timed out without the preprocessor.

	state-of-the-art		this paper				
set	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
dac (p)	0.51 ± 0.057	2.49 ± 0.072	**0.38** ± 0.006	0.91 ± 0.016	0.5 ± 0.002	0.41 ± 0.058	0.46 ± 0.005
states (p)	7.32 ± 0.031	12.55 ± 0.281	**6.79** ± 0.075	6.8 ± 0.069	6.8 ± 0.062	6.83 ± 0.085	6.8 ± 0.074
internet (p)	0.19 ± 0.005	2.47 ± 0.379	**0.14** ± 0.003	0.15 ± 0.004	**0.14** ± 0.002	**0.14** ± 0.004	**0.14** ± 0.003
ispd (p)	7.12 ± 0.069	7.04 ± 0.085	5.48 ± 0.046	**5.43** ± 0.023	5.46 ± 0.005	5.45 ± 0.025	5.45 ± 0.008
MIP2017 (p)	22.3 ± 0.22	803.63 ± 10.469	**14.07** ± 0.171	92.76 ± 2.796	28.59 ± 0.234	15.52 ± 0.324	26.54 ± 0.128
SAT2021 (p)	2217.61 ± 0.627	3645.33 ± 11.963	**1701.62** ± 10.411	1856.39 ± 10.138	1939.54 ± 4.926	1786.68 ± 6.005	1763.12 ± 6.214
SAT2021-up (p)	1886.87 ± 4.472	2948.5 ± 25.254	**1439.5** ± 2.972	1538.71 ± 8.639	1650.68 ± 3.577	1508.91 ± 2.327	1481.28 ± 3.556

Figure 5 Benchmark results on various sets of large, practical graphs (**randomly permuted**), timeout is 60*s*. The benchmarks compare solver configurations using the preprocessor ("SY+") to state of the art SAUCY and TRACES. Shown values are the sum over all instances in the set in seconds. The average and standard deviation of 3 consecutive runs is used. Bold entries indicates the fastest running time for the given set.

section. To make up **pract**, we picked the 5 largest instances (if available) of all the SAUCY benchmark sets, and for the sets arising from computational tasks (MIP and SAT) we picked 5 instances uniformly at random.

Set "comb". The goal of this set is to measure the overhead of applying the preprocessor on graphs where there is no or very little exploitable structure (i.e., where the preprocessor is expected to have no effect). For this purpose, we chose a large variety of graphs from combinatorics, on which solvers are routinely evaluated [26]. The subset we chose contains a graph from almost every graph class of the benchmark library from [2] (cfi, grid, grid-sw, had, had-sw, hypercubes, kef, latin, latin-sw, lattice, mz, paley, pp, ran10, ransq, sts, sts-sw, ranreg, tran, triang and shrunken multipedes). Whenever applicable, we chose a graph of around 1000 vertices: note that here, we apply a size restriction, since combinatorial graphs are generally difficult for their size. We choose an even smaller graph or left out sets entirely whenever a solver had trouble finishing the instance quickly. Note that, since we want to measure the preprocessing overhead, only instances for which the solvers finish in a reasonable amount of time are of interest. If solvers take a long time solving an instance to begin with, the overhead of the preprocessor is always negligible. Note that these restrictions **only apply to comb**: all the other sets tested in this paper have no restriction on the size of instances and instances were not chosen manually.

Results. The results are summarized in Figure 4. We conclude for BLISS, NAUTY and DEJAVU that the preprocessor increases performance dramatically on most instances, while the overhead of the preprocessor is negligible. For TRACES, performance also improves, in particular there are fewer timeouts. However, the improvement is not as dramatic.

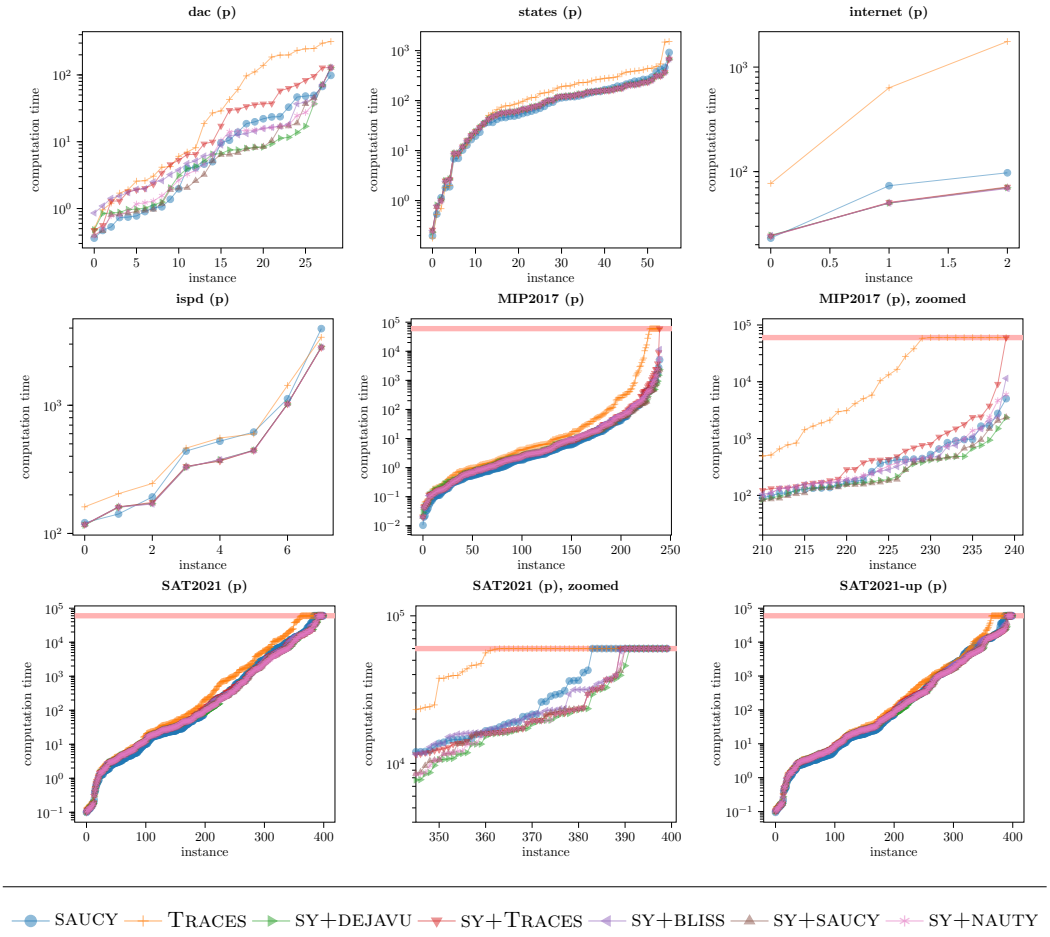


Figure 6 Detailed plots for the various sets of Figure 5. The red bar illustrates timeouts. Instances are sorted according to running time.

There are however two eye-catching instances: first, there is an instance with very high standard deviation for DEJAVU. The instance is a Kronecker eye flip graph, which DEJAVU is known to struggle with [6]. Secondly, there is a particular expensive outlier for TRACES. We analyze and discuss the instance in detail in Appendix C. There, we conclude that the outlier is caused through an undesired interaction with a heuristic of TRACES.

9.2 Comparison to state-of-the-art

The state-of-the-art solver on large practical graphs is SAUCY. Furthermore, TRACES also contains low-degree techniques. Thus, we compare all the solvers with the preprocessor to SAUCY and TRACES. The timeout used is $60s$ (also if a solver runs out of memory).

We test all sets of the SAUCY distribution. We also test 3 contemporary sets of practical graphs: the MIP2017 set contains graphs stemming from the mixed integer programming library (see [1]). The SAT2021 library contains graphs stemming from SAT instances from the SAT competition 2021 [4]. In the SAT2021-up set, SAT instances were first preprocessed using the unit and pure literal rule (see [5]). We want to remark that the SAT sets contain the largest graphs out of all the tested sets, with up to tens of millions of vertices.

The results are summarized in Figure 5, Figure 6, and Appendix B. We observe that the previous state-of-the-art (SAUCY) is outperformed on all but one set by several solvers using the preprocessor. This demonstrates that the approach of using our universal preprocessor in conjunction with different solvers can outperform state-of-the-art. Moreover, both SAUCY and TRACES also visibly speed up by applying the preprocessor on all but one set.

On a few of the very large graphs in the SAT sets, DEJAVU and TRACES run out of memory. Hence, depending on how this is weighed into the evaluation, other solvers may be preferable. In all cases where DEJAVU runs out of memory, all other solvers time out. In any case, on all these sets, SY+NAUTY and SY+SAUCY also outperform SAUCY.

10 Conclusion and Future Development

We introduced the new SASSY preprocessor for symmetry detection. We demonstrated that SASSY indeed speeds up state-of-the-art solvers on large, practical graphs. Future additions to the preprocessor could include more heuristics for degree 2 removal, stronger invariants or even more efficient implementations and tuning of the existing heuristics. Since we have observed a high sensitivity of state-of-the-art solvers to their choice of cell selectors, a more extensive study into the topic would be of interest.

References

1 MIPLIB 2017 - The Mixed Integer Programming Library. `https://https://miplib.zib.de/`.

2 nauty and Traces. `http://pallini.di.uniroma1.it`.

3 sassy. `https://github.com/markusa4/sassy`.

4 SAT Competition 2021. `https://satcompetition.github.io/2021/`.

5 Markus Anders. SAT preprocessors and symmetry. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 1:1–1:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.SAT.2022.1`.

6 Markus Anders and Pascal Schweitzer. Engineering a fast probabilistic isomorphism test. In Martin Farach-Colton and Sabine Storandt, editors, *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 73–84. SIAM, 2021. `doi:10.1137/1.9781611976472.6`.

7 Markus Anders and Pascal Schweitzer. Parallel computation of combinatorial symmetries. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ESA.2021.6`.

8 Markus Anders, Pascal Schweitzer, and Florian Wetzels. Comparative design-choice analysis of color refinement algorithms beyond the worst case. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 15:1–15:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.15`.

9 Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. `doi:10.1007/s00224-016-9686-0`.

10 Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In Nikolaj S. 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 101–115. Springer, 2011. `doi:10.1007/978-3-642-22438-6_10`.

11 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, pages 530–534. ACM, 2004. doi:10.1145/996566.996712.

12 David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In Nikolaj S. 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. doi:10.1007/978-3-642-22438-6_18.

13 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2017. doi:10.1007/978-3-319-66263-3_6.

14 Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. doi:10.1007/11499107_5.

15 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006. doi:10.1016/S1574-6526(06)80014-3.

16 Mark K. Goldberg. A nonfactorial algorithm for testing isomorphism of two graphs. *Discret. Appl. Math.*, 6(3):229–236, 1983. doi:10.1016/0166-218X(83)90078-1.

17 Christopher Hojny and Marc E. Pfetsch. Symmetry handling via symmetry breaking polytopes. In Ekrem Duman and Ali Fuat Alkaya, editors, *13th Cologne Twente Workshop on Graphs and Combinatorial Optimization, Istanbul, Turkey, May 26-28, 2015*, pages 63–66, 2015.

18 Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of Computational Group Theory*. Discrete Mathematics and Its Applications. Chapman and Hall/CRC, 2005.

19 Tommi A. Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In Alberto Marchetti-Spaccamela and Michael Segal, editors, *Theory and Practice of Algorithms in (Computer) Systems – First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings*, volume 6595 of *Lecture Notes in Computer Science*, pages 151–162. Springer, 2011. doi:10.1007/978-3-642-19754-3_16.

20 Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Symmetry and satisfiability: An update. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2010. doi:10.1007/978-3-642-14186-7_11.

21 Manuel Kauers and Martina Seidl. Symmetries of quantified boolean formulas. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing – SAT 2018 – 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2018. doi:10.1007/978-3-319-94144-8_13.

22 Sandra Kiefer, Ilia Ponomarenko, and Pascal Schweitzer. The weisfeiler-leman dimension of planar graphs is at most 3. *J. ACM*, 66(6):44:1–44:31, 2019. doi:10.1145/3333003.

23 Sandra Kiefer, Pascal Schweitzer, and Erkal Selman. Graphs identified by logics with counting. *ACM Trans. Comput. Log.*, 23(1):1:1–1:31, 2022. doi:10.1145/3417515.

24 Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: An extended maxsat preprocessor. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017 – 20th International Conference, Melbourne, VIC, Australia, August 28 – September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 449–456. Springer, 2017. doi:10.1007/978-3-319-66263-3_28.

25 François Margot. Symmetry in integer linear programming. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008 – From the Early Years to the State-of-the-Art*, pages 647–686. Springer, 2010. doi:10.1007/978-3-540-68279-0_17.

26 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

27 Marc E. Pfetsch and Thomas Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Math. Program. Comput.*, 11(1):37–93, 2019. doi:10.1007/s12532-018-0140-y.

28 Adolfo Piperno. Isomorphism test for digraphs with weighted edges. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy*, volume 103 of *LIPIcs*, pages 30:1–30:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.SEA.2018.30.

29 Pascal Schweitzer and Daniel Wiebking. A unifying method for the design of algorithms canonizing combinatorial objects. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1247–1258. ACM, 2019. doi:10.1145/3313276.3316338.

30 Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003. doi:10.1017/CBO9780511546549.

A Probing for Sparse Automorphisms

A.1 The Individualization Refinement Framework

The individualization-refinement framework is a general framework for algorithms computing isomorphisms, automorphisms and canonical labellings (see [26]). These algorithms generally work on a special tree, the so-called *IR tree*. We give a brief description of how IR trees are constructed. For a more extensive description, in particular for the numerous strategies needed to perform the search more efficiently, see [26, 7].

Each node x of the tree has a corresponding equitable coloring π_x of the input graph. The leaves correspond to discrete colorings. The most important property of IR trees is that they are isomorphism-invariant, meaning that on G and $\varphi(G)$ (where φ is an isomorphism) we obtain isomorphic IR trees.

Let (G, π) be the input graph. Let π' be the coarsest equitable refinement of π. We let the root of the IR tree correspond to π'. In each node x of an IR tree, a non-trivial color class from the corresponding coloring π_x is chosen (i.e., a $C = \pi_x^{-1}(c)$ with $|C| > 1$, C must be chosen isomorphism-invariantly). If there is no non-trivial color class, then x is a leaf and its corresponding coloring is discrete. Otherwise, for each $v \in C$, we define x_v as a child of x in the IR tree. Let π_{x_v} denote the coloring corresponding to x_v. We may obtain π_{x_v} from π_x as follows. Starting from π_x, we first artificially single out v (i.e., *individualize* v). This means we set $\pi_{x_v}(v) := c'$ where $c' \notin \pi(V(G))$ (again, c' is chosen isomorphism-invariantly). Then, we refine the coloring using color refinement, obtaining the equitable coloring π_{x_v}.

We can derive automorphisms from IR trees. If π_1, π_2 are leaves of the tree, i.e., discrete colorings, then $\varphi := \pi_1^{-1} \circ \pi_2$ defines a permutation on $V(G)$. While φ is not guaranteed to be an automorphism, we can efficiently test whether it is (by checking whether $\varphi(G) = G$).

With this method all of $\mathrm{Aut}(G)$ can be computed. This follows essentially from the fact that comparing all pairs of leaves in this way will give us all automorphisms of G (or rather a generating set of $\mathrm{Aut}(G)$ when automorphism pruning is applied; see [26]).

A.2 The Probing Algorithm

Our probing strategy only searches for automorphisms which can be used directly to reduce the graph. The idea is as follows. For a color class that we want to reduce, we attempt to collect automorphisms that transitively permute all the vertices in the entire color class. This certifies that the color class is an orbit. We can then individualize an arbitrary vertex of the color class. In contrast, if we only have some automorphisms that together do not act transitively on the color class, it is not clear how to manipulate the graph favorably. In particular, since some of the vertices may not be in the same orbit, we do not know which vertex to individualize. We now describe the bounded IR probing algorithm.

(Description of Algorithm 1.) (See Algorithm 1 for the pseudocode.) The algorithm expects as input a colored graph $G = (V, E, \pi)$, a color class $C_{probe} = \pi^{-1}(c)$ as well as a length bound L. It outputs a set of automorphisms Φ and a coloring π' refining π. If the probing was unsuccessful then $\Phi = \{\}$ and $\pi' = \pi$. Otherwise $\langle\Phi\rangle$ acts transitively on C_{probe} and π' is obtained from π by individualizing a vertex and refining.

We compute arbitrary IR paths (i.e., a rooted path in the IR tree) starting with an individualization of a vertex in C_{probe}. The path is only computed up to a length of L.

Initially, the algorithm examines two of these paths concurrently, starting in two different vertices $v_1, v_2 \in C_{probe}$. It checks after each individualization whether the automorphism φ_{π_1,π_2} (defined, as above, mapping corresponding singletons) is an automorphism. If this happens to be the case after having performed, say, L' individualizations, we bound all subsequent paths by L'.

Afterwards for each vertex $w \in C_{probe} \setminus \{v_1, v_2\}$ we compute an IR path starting with the individualization of w. We hope to find an automorphism mapping v_1 to w. If we discover an automorphism for each w, we return the set of automorphisms Φ, individualize v_1 in π, refine to obtain π' and return Φ and π'.

(Correctness of Algorithm 1.) Correctness of the algorithm follows simply from the fact that we certify all automorphisms. That is, every map claimed to be an automorphism is indeed an automorphism. Since this certification is done for each automorphism, this certifies the fact that C_{probe} is an orbit of $\mathrm{Aut}(G, \pi)$. Since we return all automorphisms required to construct the orbit (i.e., we return Φ), we have $\langle\Phi \cup \mathrm{Aut}(G, \pi')\rangle = \mathrm{Aut}(G, \pi)$ by the orbit-stabilizer theorem (see [18]).

(Implementation of Algorithm 1.) We want to make some further remarks on the implementation of the algorithm. In fact, even though it can be implemented very efficiently, it generally has to be used sparingly. Overall we need to decide when and how often to employ the probing strategy and also which depth bound L to use. The preprocessor essentially uses three strategies: 1-IR probing, ∞-IR probing with class size 2 and ∞-IR probing up to class size 8 (in order of descending frequency).

B Non-permuted Benchmarks

Figure 7, Figure 8 and Figure 9 show benchmark results for *non-permuted* graphs. While overall times are faster across all graph classes and solvers than on the randomly permuted graphs, the interpretation of results given in Section 9 also applies to these benchmarks. Hence, our results agree on both randomly permuted and non-permuted graphs.

Algorithm 1 Bounded IR probing in a color class C_{probe} up to a path of length L.

```
1  function BoundedProbeIR(G, π, C_probe, L)
       Input  : graph G = (V, E, π) where π is equitable, color class C_probe of π, length
                bound L
       Output : (equitable) coloring π', set of automorphisms Φ
2      Φ := {} ;                                              // set of automorphisms
3      Pick vertices v_1, v_2 ∈ π^{-1}(C_probe);
4      for i ∈ {1, 2} do
5          π_i := π;
6          individualize v_i in π_i;
7          ColorRefinement(G, π_i);
8      L_C := [C_probe] ;                                     // list of color classes
9      while |L_C| < L do
10         if φ_{π_1,π_2}(G, π) = (G, π) then
11             break;                                         // automorphism found
12         C := non-trivial color class of π_1;
13         L_C += [C] ;                                       // append C to L_C
14         individualize some v ∈ π_1^{-1}(C) in π_1;
15         ColorRefinement(G, π_1);
16         individualize some v ∈ π_2^{-1}(C) in π_2;
17         ColorRefinement(G, π_2);
18     if φ_{π_1,π_2}(G, π) ≠ (G, π) then
19         return π, ∅;                                       // probing failed
20     else
21         Φ := Φ ∪ {φ};
22     for w ∈ C_probe \ {v_1, v_2} do
23         reset π_2 to π ;                                   // essentially π_2 := π
24         individualize w in π_2;
25         ColorRefinement(G, π_2);
26         for C ∈ L_C do
27             individualize some v ∈ π_2^{-1}(C) in π_2;
28             ColorRefinement(G, π_2);
29         if φ_{π_1,π_2}(G, π) ≠ (G, π) then
30             return π, ∅;                                   // probing failed
31         else
32             Φ := Φ ∪ {φ};
33     individualize v_1 in π ;                // success; individualize v_1 in (G, π)
34     return ColorRefinement(G, π), Φ;
```

C The Outlier in Combinatorial Graphs

There is one particular outlier in the evaluation of Traces comparing preprocessed vs. unprocessed instances. The instance is a shrunken multipede on 408 vertices. Without preprocessing, it is solved in $0.75s$, while with preprocessing it is solved in $6.3s$. This is at first glance confusing: the preprocessor finishes within less than 0.5ms, while not changing the graph other than coloring it with its coarsest equitable coloring. This is however almost the same coloring Traces would also compute for the graph.

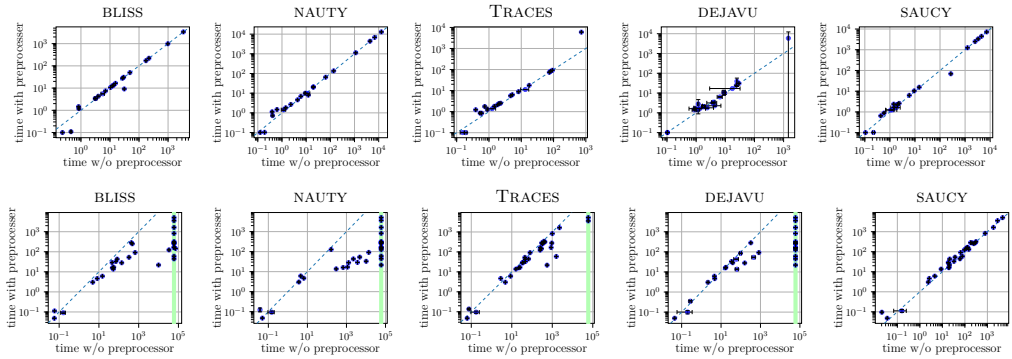


Figure 7 Solvers with SASSY vs. solvers without SASSY on **comb** (top) and **pract** (bottom), not permuted. Timeout is 60s. The green bar shows instances that timed out without the preprocessor.

	state-of-the-art		this paper				
set	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
dac	0.35 ± 0.008	2.47 ± 0.005	0.28 ± 0.001	0.81 ± 0.002	0.37 ± 0.002	**0.27** ± 0.002	0.34 ± 0.001
states	**2.89** ± 0.054	7.58 ± 0.158	3.85 ± 0.048	3.85 ± 0.04	3.85 ± 0.041	3.85 ± 0.038	3.85 ± 0.043
internet	0.15 ± 0.002	2.23 ± 0.022	**0.13** ± 0.000	**0.13** ± 0.001	**0.13** ± 0.001	**0.13** ± 0.000	**0.13** ± 0.000
ispd	3.83 ± 0.028	4.84 ± 0.059	3.7 ± 0.057	3.7 ± 0.061	3.7 ± 0.057	3.7 ± 0.054	**3.68** ± 0.039
MIP2017	10.96 ± 0.158	774.09 ± 0.578	**9.12** ± 0.165	84.42 ± 0.201	21.46 ± 0.331	10.66 ± 0.109	21.04 ± 0.163
SAT2021	1292.76 ± 1.641	2855.57 ± 10.636	**881.69** ± 0.982	1058.97 ± 3.283	1149.04 ± 3.73	990.96 ± 3.323	988.15 ± 2.662
SAT2021-up	1144.06 ± 3.648	2393.85 ± 4.799	**780.02** ± 6.009	903.93 ± 2.517	1027.61 ± 3.815	876.77 ± 2.535	865.38 ± 2.578

Figure 8 Benchmark results on various sets of large, practical graphs (**not permuted**), timeout is 60s. Running out of memory also counts as a timeout. The benchmarks compare solver configurations using the preprocessor ("SY+") to state of the art SAUCY and TRACES. Shown values are the sum over all instances in the set in seconds. The average and standard deviation of 3 consecutive runs is used. Bold entries indicates the fastest running time for the given set.

The only difference is that TRACES might name the colors differently internally, e.g., color 3 might be named color 6 instead. While this does not structurally make the graph harder or easier, heuristics internally might always, for example, choose the "first largest color" (this is similar to, e.g., variable ordering in SAT solvers). Thus, renaming the colors might influence the decisions made by the solver. Using the "first" color is however usually not a deliberate decision. In fact, if we simply reverse the order of colors, the graph is indeed solved in 0.12s. In [7], it is also argued that cell selector choice has a significant impact on the set of shrunken multipedes. We believe that the solution to this issue is to make structurally better choices, and has indeed little to do with the role of the preprocessor.

D Ablation study

In Figure 10 we evaluate for DEJAVU on the MIP2017 set the effect of each of the preprocessing techniques separately. We do so by running the configuration SY+DEJAVU, but performing a separate run for each technique, deactivating the respective technique. For example, SY+DEJAVU-DEG2 runs SY+DEJAVU without the degree 2 removal techniques. The data shows that each of the techniques has a beneficial impact on running time. By far the most impactful technique is the removal of degree 0 and 1, followed by the removal of vertices of degree 2, and lastly probing.

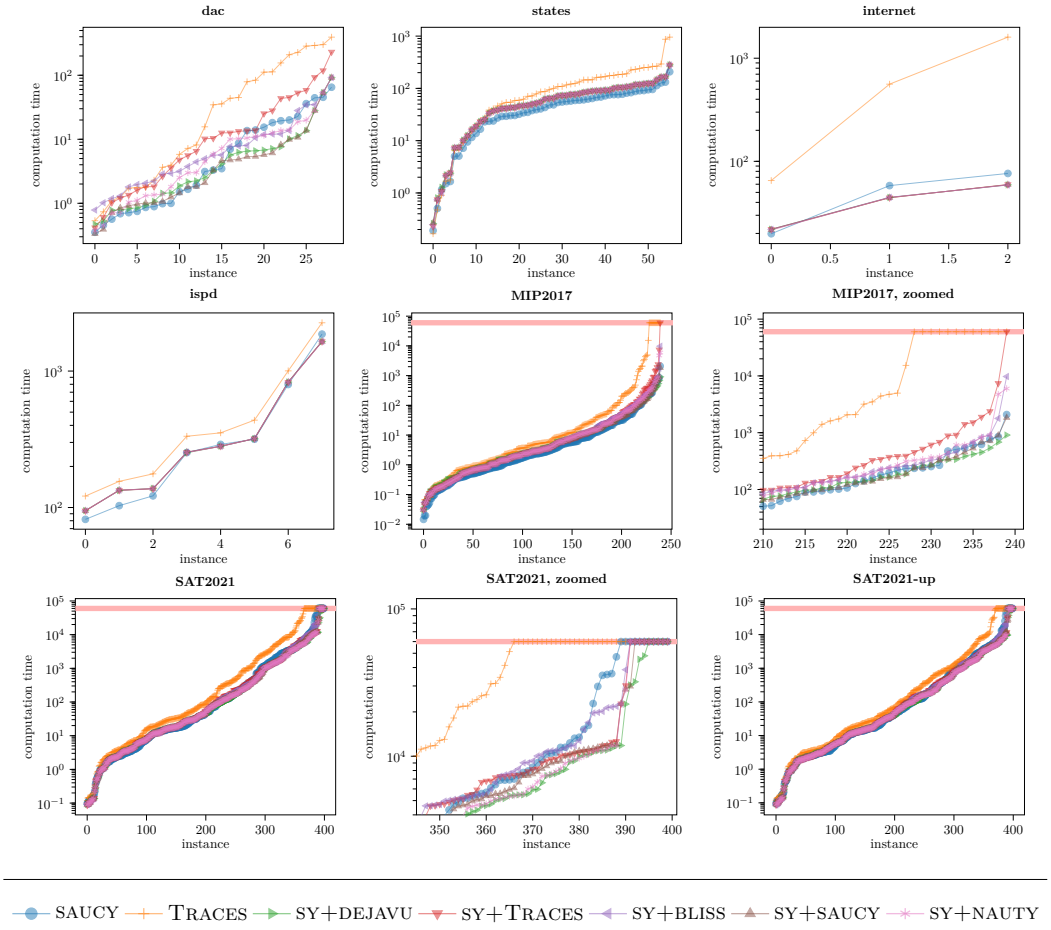


Figure 9 Detailed plots for the various sets of Figure 8. The red bar illustrates timeouts. Instances are sorted according to running time.

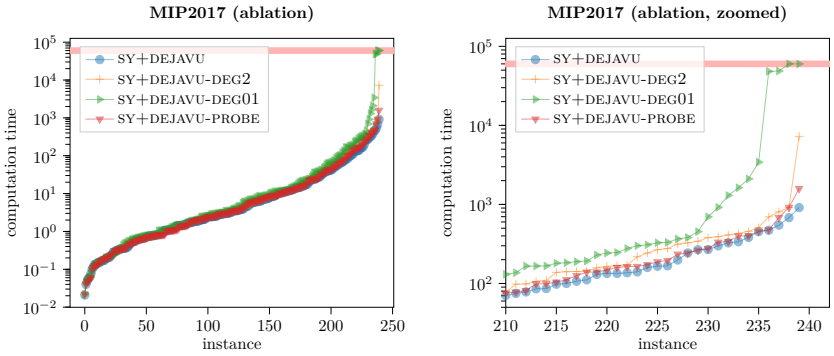


Figure 10 Ablation study for SY+DEJAVU on the MIP2017 graphs (times: 9.15s, 17.87s, 234.47s, 10.65s), timeout is 60s.