Symmetry-Preserving Graph Compression

Markus Anders
RPTU Kaiserslautern-Landau, Germany

Manuel Penschuck
University of Southern Denmark, Odense, Denmark

Pascal Schweitzer
TU Darmstadt, Germany

Abstract

Exploiting symmetry is a well-established technique to eliminate redundant work in combinatorial solvers, yet it often incurs computational overhead that limits its practical impact. In particular, practical instances arising from applications are frequently large and give rise to graphs whose size becomes a major bottleneck for algorithms dealing with symmetry.

We propose a method for symmetry-preserving graph compression that reduces graph size while preserving the symmetries of the original graph in a controlled way. Our approach identifies and merges equivalent vertex colors under conditions that guarantee the recoverability of all symmetries. We provide both a theoretical foundation and efficient practical criteria for such merges, show that computing optimal and approximately optimal compression is intractable, and introduce a linear-time, practical heuristic. Extensive experiments on a vast library of graphs demonstrate that our new technique achieves significant compression ratios. Implemented in the state-of-the-art symmetry detection tool DEJAVU, we achieve an overall speedup of 1.39, with large modern SAT and MIP benchmarks benefiting the most.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases symmetry detection, graph automorphisms, preprocessing, color refinement, approximation hardness, mathematical programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.7

Supplementary Material *Software (maintained version)*: `https://github.com/markusa4/dejavu`
Software (archived version): `https://zenodo.org/records/21158722`

Funding This work was partially funded by the Novo Nordisk Foundation grant NNF21OC0066551.

Acknowledgements We thank Christopher Hojny for providing the MIPLIB graphs used in our experiments. A substantial part of the computation for this project was performed on the UCloud interactive HPC system managed by the eScience Center at the University of Southern Denmark.

1 Introduction

Exploiting symmetries is a standard technique in a wide range of combinatorial optimization and constraint-solving paradigms, such as mixed-integer programming [14] (MIP), Boolean satisfiability [2] (SAT), satisfiability modulo theories [9] (SMT), pseudo-Boolean solving [22] (PB), and constraint programming [12] (CP). Redundant search can be avoided by, for example, introducing symmetry-breaking constraints [6], fixing variables without changing the optimal solution value [18], using symmetry-aware branching rules [23], or learning symmetric explanations for the problem at hand [10].

However, independent of how exactly we want to exploit symmetries, we need a way to detect them efficiently first. In order to do so, computational problems are usually modelled as graphs, such that the symmetries of the problem are precisely the symmetries of the

graph. Indeed, symmetry detection on graphs constitutes a core subroutine in most modern combinatorial solvers. Practical symmetry detection algorithms such as NAUTY [20, 21], TRACES [25, 21], SAUCY [7, 8], BLISS [16, 17], and DEJAVU [3, 1] have been developed over more than 50 years. They are all based on the individualization-refinement (IR) paradigm [24], but differ in their search strategies and heuristics.

Often, symmetry is used to accelerate another solver. Thus, all symmetry-related computations need to be very fast to avoid incurring excessive overhead, which would lead to an overall slowdown rather than speedup. A particular challenge arises from practical instances, for example stemming from SAT or MIP problems: these instances are often very large, and already for this reason can significantly slow down the underlying computations.

Recently, Anders, Schweitzer, and Stieß [4] proposed to simplify the input graph while preserving its automorphism group in a controlled manner, with the aim of obtaining a smaller instance on which symmetry detection can be performed more efficiently. Experiments demonstrate that such simplifications can yield significant speedups on large practical instances [4, 1]. Importantly, these ideas have since been adopted in practice: they are used in the MIP solver SCIP [14], as well as in the SAT symmetry breaking tool SATSUMA [2].

This naturally raises the question of whether such size reductions can be developed more systematically and what their limits are. Do practical instances admit redundant structures that are not essential for representing their symmetries, and can these be removed in a principled way? More specifically, can the graph modelling the symmetries of a given instance be *compressed* into a more compact representation without losing information? Crucially, this requires us to understand when parts of the graph can be merged with other parts of the graph without changing its automorphism group. The resulting compact representation of the graph could benefit not only symmetry detection, but more generally any downstream algorithm operating on the same graph.

Indeed, the need for effective symmetry-preserving graph compression is becoming more evident. To control cost, most modern combinatorial solvers invoke symmetry techniques only under tight resource limits [2, 14]. On large instances they are often disabled entirely. As a result, the use of symmetry in combinatorial solvers is limited in practice, restricting its practical benefits despite the strong theoretical potential [28].

More powerful symmetry-handling techniques have long been known in the literature, yet their practical adoption remains limited due to their prohibitive computational overhead, especially at scale. For example, a very recent and promising approach to deal with symmetry more effectively is to introduce *iteration* [15]. Rather than computing symmetries only *once*, this approach computes symmetries *repeatedly* within a preprocessing loop. This change has important implications: the cost of symmetry handling is no longer a one-time expense, but accumulates over several iterations of the procedure. Therefore, the efficiency of algorithms dealing with symmetry becomes increasingly critical. Even more ambitious approaches, such as local symmetry handling [11], conditional symmetry handling [11], or isomorphism pruning [19] use symmetry detection and related routines even more frequently.

In summary, while symmetry handling algorithms have been highly optimized over decades, the size of current practical input graphs remains a crucial obstacle. This motivates the use of graph compression: by shrinking instances upfront, one can directly address the root cause of the computational bottleneck when working with large practical instances.

Contributions. We propose a new method for *symmetry-preserving graph compression*, targeting large practical instances from SAT and MIP. Our approach reduces the size of a graph by efficiently merging *equivalent colors*, thereby obtaining a more compact representation while preserving its automorphism group.

Vertex colors dictate that only vertices of the same color can be mapped onto each other by symmetries. Computing and refining such colorings is central to all symmetry detection algorithms. We study the case where two colors are *equivalent*, meaning that the action of any symmetry on one color uniquely determines its action on the other. This suggests that such colors can potentially be merged, yielding a smaller graph representation. However, performing such merges naively may alter the symmetries in an irrecoverable way.

Our main goals are therefore to identify conditions under which merging colors is possible in theory, to understand its computational complexity, and to develop efficient algorithms for its practical application. Our contributions are as follows:

- We describe general local conditions (mergeability) under which equivalent colors of a given graph can safely be merged, and the original symmetries recovered.
- Then, we give less general but sufficient conditions (easy mergeability) for merging colors that can be checked efficiently and easily in practice.
- We prove an $n^{1/3-\epsilon}$ approximation hardness result for deriving the *optimal* sequence of merges, both using mergeability and easy mergeability. In other words, optimal or close-to-optimal application of local merges is computationally intractable.
- We describe a greedy linear-time heuristic which merges easily mergeable colors.

Benchmarks on a vast set of graphs show that the new algorithm achieves strong compression ratios on practical instances. Excluding instances already reduced to zero by trivial preprocessing, the total size of benchmarks is reduced to 11.7% of the size achieved by previous methods. Integrated into the state-of-the-art symmetry detection tool DEJAVU this yields a speedup of 1.39. In both cases, large modern SAT and MIP benchmarks benefit the most.

2 Preliminaries

We begin by introducing general concepts and notation for graphs and symmetries.

Graphs. We consider finite, simple, undirected graphs $G = (V, E)$ with $V = \{0, \dots, n-1\}$ and $E \subseteq \binom{V}{2}$. We write $n = |V|$, $m = |E|$, and denote the size of G by $n+m$. For convenience, we use $V(G)$ and $E(G)$ to refer to the vertex and edge sets of G, respectively. For $V' \subseteq V(G)$, the *induced subgraph* $G[V']$ is defined by $V(G[V']) = V'$ and $E(G[V']) = E(G) \cap \binom{V'}{2}$. For two sets V_1 and V_2, we define $V_1 \boxtimes V_2 \coloneqq \{\{v_1, v_2\} : v_1 \in V_1,\ v_2 \in V_2,\ v_1 \neq v_2\}$, which forms the set of distinct pairs of the Cartesian product and turns them into 2-sets.

A *vertex coloring* of a graph $G = (V, E)$ is a map $\pi \colon V \to \{0, \dots, n-1\}$. For $i \in \{0, \dots, n-1\}$ with $\pi^{-1}(i) \neq \emptyset$, the set $\pi^{-1}(i) \subseteq V$ is called a *color class*. A *vertex-colored graph* is a tuple (G, π). Unless stated otherwise, the graphs referred to in this paper are vertex-colored graphs. For a map f we denote by $f|_D$ the restriction of f to a subset D of the domain. If G is a colored graph and V' a subset of the vertices then the induced graph $G[V']$ inherits the coloring $\pi|_{V'}$, i.e., vertices always keep their color. In practice, even when the input graph is uncolored, all competitive graph isomorphism solvers compute and refine vertex colorings using the color refinement algorithm [21].

Isomorphisms. An *isomorphism* between uncolored graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is a bijection $\varphi \colon V_1 \to V_2$ such that $(u, v) \in E_1$ if and only if $(u^\varphi, v^\varphi) \in E_2$. Two *vertex-colored* graphs (G_1, π_1) and (G_2, π_2) are *isomorphic* if there exists an isomorphism $\varphi \colon V_1 \to V_2$ such that $\pi_1(v) = \pi_2(\varphi(v))$ for all $v \in V_1$. An *automorphism* of G is an isomorphism from the graph to itself. We denote by $\mathrm{Aut}(G)$ the set of all automorphisms of a graph G. This set forms a permutation group. The *orbit* of $v \in V$ under $\mathrm{Aut}(G)$ is $\mathrm{Orb}(v) \coloneqq \{\varphi(v) \mid \varphi \in \mathrm{Aut}(G)\}$.

3 Equivalent Orbits

Intuitively, in a permutation group, two orbits are equivalent if there is a one-to-one correspondence between the elements of the two orbits so that the action on one orbit dictates the action on the other orbit via the correspondence, and vice versa.

▶ **Definition 1.** *Two orbits O_1, O_2 of* $\mathrm{Aut}(G)$ *are called* equivalent *if there exists a bijection $M\colon O_1 \to O_2$ such that for every $\varphi \in \mathrm{Aut}(G)$ and every $v \in O_1$, $M(\varphi(v)) = \varphi(M(v))$.*

Thus, if O_1 and O_2 are equivalent, they are linked by a bijection preserved by all automorphisms. In particular, moving a vertex to a particular target in one orbit forces the movement of the corresponding vertex to the corresponding target in the other orbit. We should remark that the bijection M does not have to be unique.

The notion of equivalent orbits plays a vital role in centralizer computations [27]. In fact, two orbits are equivalent exactly if there is a centralizer element (in the symmetric group) interchanging the orbits. The theory shows that, using point-stabilizers, it is algorithmically readily feasible to compute all equivalent orbits and the corresponding bijections:

▶ **Lemma 2** ([27, Lemma 6.1.9.])**.** *Two orbits O_1 and O_2 are equivalent if and only if they have the same size and in the stabilizer* $\mathrm{Aut}(G)_\alpha$ *of some (and thus every) point $\alpha \in O_1$ some point of O_2 is fixed.*

In our context, the following observation is important. If two orbits O_1 and O_2 are equivalent, then the group induced on $V \setminus O_2$ is isomorphic to $\mathrm{Aut}(G)$. This means that, given M, by considering only the symmetries $\mathrm{Aut}(G)|_{V\setminus O_2}$, we do not lose any information. Indeed, if we know the bijection we can reconstruct the full group.

Our goal is to reduce graphs by merging equivalent orbits. In this situation, we may remove, say, O_2 and somehow redirect all its adjacencies to O_1. If the automorphism group is preserved under this operation, automorphisms of the reduced graph can be lifted canonically to automorphisms of the original graph via the bijection.

In our context, however, there is a major caveat to this approach. Since we are still in the process of computing the automorphism group of the graph, we do not have access to the group. This means, we need to perform the merge without actually knowing the entire group. We also have to encode our result into a graph since using group-theoretic machinery would be computationally too expensive. Consequently, two major challenges arise.

1. Under which theoretical conditions can we construct a new graph to capture all of the automorphism group information after the merge?
2. How can we detect in practice that two orbits are equivalent without knowing the group and construct a compressed graph?

The first question is more of a structural nature, and we will address it in this and the next section. The second question is algorithmic and will be discussed in Sections 5 and 6.

In general, not every permutation group is the automorphism group of a graph. In fact, there are three major obstacles to a permutation group being a graph's automorphism group:

- A group is called 2-*closed* if it is the automorphism group of a binary relational structure. Since graphs are binary relational structures, only 2-closed groups can be automorphism groups of graphs. However, not every permutation group is 2-closed: for example, the alternating group in its natural action is not 2-closed. Hence not every permutation group can be realized as the automorphism group of a graph

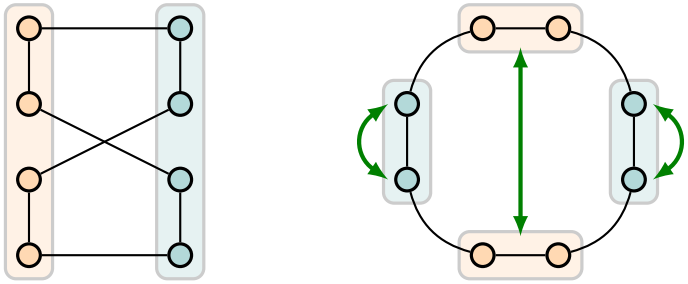

Figure 1 The 8-cycle partitioned into two colors. On the right, the automorphisms are illustrated: exchanging the edges in one color requires us to reverse the endpoints of the edges in the other. The induced actions on the blue or orange color cannot be expressed using a graph on 4 vertices.

- Our graphs are undirected, but a permutation group may need a *directed* relation to be represented as an automorphism group. An example is the cyclic group $\mathbb{Z}_3$ in natural action.
- Finally, to represent a group we may need more than one binary relation. In our concrete application scenario this is a severe restriction, which is due to the fact that current efficient tools for symmetry handling do not allow the use of colored edges[1]. It turns out that even if two groups are given as automorphism groups of undirected graphs, when they are joined via a matching in a larger graph with two color classes, the resulting group on each of the classes may require more than one relation. We discuss an example of this phenomenon next.

▶ **Example 3.** Consider the 8-cycle C_8 and color the vertices into two sets of colors so that each color class induces $2K_2$, the disjoint union of two cliques of size two (see Figure 1). The automorphism group of this graph is isomorphic to $\mathbb{Z}_2^2$. It acts faithfully on both color classes, which form equivalent orbits. Considering the list of transitive graphs on 4 vertices (K_4, $2K_2$ and their complements) shows that the group cannot be represented as the automorphism group of a graph on 4 vertices. (We should highlight that allowing directed edges does not mitigate the issue.)

The example can be amplified by taking many disjoint copies of the graph just described. This gives us an infinite family of graphs with two equivalent orbits such that the permutation group induced on each orbit is not representable as automorphism group of a graph. In fact, if we try to approximate the group as automorphism group of a graph, we will see that every over- or under approximation will yield supergroups or subgroups of unbounded index.

Consequently, in our context of undirected graphs without edge colors, we will not be able to fully reduce equivalent orbits. We will have to be content with some partial compression.

4 Merging Colors in Graphs

In our situation, we do not know the orbits, since they essentially are what we are trying to compute in the first place. We will thus generalize the notion of equivalent orbits to colors. Recall that symmetries need to respect colors and thus color classes are unions of orbits. Our new goal is to reduce graphs by merging equivalent color classes.

[1] With the notable exception of an experimental version of Traces called WTraces [26].

Hence, the key challenge is to efficiently detect when two color classes are equivalent in this sense, and when we can construct a new merged graph that preserves all automorphisms. In this section, we give general sufficient conditions for the soundness of such merge operations, which will serve as a correctness foundation for the more practical criteria introduced later.

4.1 Mergeability

Equivalence-ensuring Bijection. Let C_1 and C_2 be two color classes of G. We say that a bijection $M\colon C_1 \to C_2$ is *equivalence-ensuring* if it is respected by all automorphisms of G (sometimes called $\mathrm{Aut}(G)$-equivariant), i.e., for all $\varphi \in \mathrm{Aut}(G)$ and all $v \in C_1$, it holds that $M(\varphi(v)) = \varphi(M(v))$. Intuitively, this means that M certifies that C_1 and C_2 are composed of pairs of equivalent orbits in $\mathrm{Aut}(G)$. We will assume that such a bijection is given. Two concrete examples when this is the case would be when the edges with one endpoint in C_1 and one endpoint in C_2 form a matching (see, e.g., Figure 1). Subdividing all these edges, with the midpoints forming a third color class, gives a second example.

Given a set of permutations Θ that stabilize C_1 and C_2 as sets, we define $\mathrm{Stab}_M(\Theta) \coloneqq \{\varphi \in \Theta \mid \forall v \in C_1 : M(\varphi(v)) = \varphi(M(v))\}$ as the subset of those permutations that respect M.

Mergeability. Assume we are given two color classes C_1 and C_2 together with an equivalence-ensuring bijection $M\colon C_1 \to C_2$. We give sufficient conditions under which we can remove C_2 from G and redirect the connections of C_2 with adjacent color classes to C_1 in such a way that we can recover the original automorphisms of G by lifting from C_1 to C_2. The precise definition for removing colors and lifting automorphisms will be given later.

Let $\mathcal{D}$ be the set of color classes that are neighbors of C_2 but not in $\{C_1, C_2\}$. Let $V_{\mathcal{D}} \coloneqq \bigcup_{D\in\mathcal{D}} D$ be the vertices in the classes in $\mathcal{D}$. Let $\widetilde{G}$ be the graph obtained from $G[V_{\mathcal{D}} \cup C_1 \cup C_2]$ by deleting all edges that do not have at least one endpoint in C_1 or C_2.

For a graph G with color classes C_1 and C_2 and an equivalence-ensuring bijection M, a graph G' is a *suitable replacement graph* (w.r.t. C_1, C_2 and M) if the following hold:

- $V(G') = V_{\mathcal{D}} \cup C_1$,
- $\mathrm{Aut}(G') = \mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))|_{V_{\mathcal{D}}\cup C_1}$,
- $E(G') \cap D_1 \boxtimes D_2 = \emptyset$ for $D_1, D_2 \in \mathcal{D}$.

Note that the last property also applies when $D_1 = D_2$. We call two color classes *mergeable* if there is an equivalence-ensuring bijection M and a corresponding suitable replacement graph G'.

Intuitively, the suitable replacement graph consists only of internal edges of C_1, as well as edges between C_1 and color classes in $\mathcal{D}$. In particular, it contains no edges between vertices of $V_{\mathcal{D}}$. Its automorphism group must exactly capture, on the vertex set $V_{\mathcal{D}} \cup C_1$, those automorphisms of the local graph $\widetilde{G}$ that respect M.

Merging Colors. We now define precisely what it means to merge two mergeable colors. Given a graph G with color classes C_1, C_2 equivalent via an equivalence-ensuring bijection M and a suitable replacement graph G', we describe the *reduced graph* $G_{C_1 \approx_M C_2}$. (The replacement graph G' missing in the notation will always be clear from context.)

The reduced graph $G_{C_1\approx_M C_2}$ has vertex set $V(G) \setminus C_2$. We start with the graph $G - C_2$ obtained from G by deleting the vertices of C_2 and all edges incident with them. From this graph, we then delete precisely those edges $\{v_1, v_2\}$ with $v_1 \in C_1$ and $v_2 \in C_1 \cup \mathcal{D}$. We then add all edges in $E(G')$. In other words, $G_{C_1\approx_M C_2}$ agrees with G' where possible and with G everywhere else.

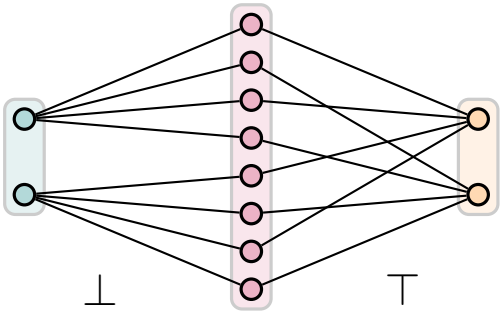

Figure 2 The blocking gadget used for the approximation hardness reduction.

4.2 Merging and Lifting

Lifting Automorphisms. Next, we describe how the automorphisms of the reduced graph $G_{C_1 \approx_M C_2}$ are lifted back to automorphisms of the original graph. Let M denote the mapping that was used in the reduction. For an automorphism $\varphi \in \mathrm{Aut}(G_{C_1 \approx_M C_2})$ we define the lifted map, a permutation of $V(G)$ which will turn out to be an automorphism, via

$$\varphi{\uparrow_M}(v) := \begin{cases} \varphi(v) & \text{if } v \notin C_2 \\ M(\varphi(M^{-1}(v))) & \text{if } v \in C_2. \end{cases}$$

This lifting procedure can be seen as a special case of the canonical strings described in previous work [4]. For a set of automorphisms Θ we define the set of lifts $\Theta{\uparrow_M} := \{\varphi{\uparrow_M} \mid \varphi \in \Theta\}$.

Mergeability is Sufficient. We now argue that mergeability is sufficient. Specifically, we argue that we can recover all automorphisms of the original graph from the reduced graph.

▶ **Theorem 4.** *Let G be a graph, M an equivalence-ensuring bijection for color classes C_1, C_2, and G' a suitable replacement graph. Then* $\mathrm{Aut}(G) = \mathrm{Aut}(G_{C_1 \approx_M C_2}){\uparrow_M}$.

Proof sketch, full proof in Appendix A. The edge relation of the suitable replacement graph suffices to reproduce the original automorphism group, when automorphisms are lifted. ◀

5 Hardness of Optimal Reduction

Naturally, we want to merge as many colors as possible, that is, reduce the graph as much as possible. We investigate the problem of optimally reducing a graph by merging colors.

▶ **Problem 1** (Optimal Merges)**.** Given a graph G, compute a graph G' of smallest order obtainable from G by merging colors according to Section 4.1.

We want to prove that optimally merging colors is computationally hard. In order to do so, we require the following gadget construction.

Blocking gadget. We introduce the *blocking gadget*. The gadget is used to connect color classes of size 2. Size-2 color classes can be connected in two different ways to the gadget, the $\bot$-connection and the $\top$-connection. The goal is that, whenever two color classes are connected to the gadget using *different* connection types, one as $\bot$ and one as $\top$, then they *cannot* be merged. In other words, the merge is *blocked*. Furthermore, the gadget itself cannot be merged with any other part of the graph, and the connection type cannot be altered by merge operations.

Figure 2 illustrates the blocking gadget. The gadget is a color $D = \{d_1, \dots, d_8\}$ with 8 vertices. Connecting a color $A = \{a_1, a_2\}$ using a $\bot$-connection to D means we connect $\{d_1, d_2, d_3, d_4\}$ to a_1 and $\{d_5, d_6, d_7, d_8\}$ to a_2 or vice versa. Connecting a color $B = \{b_1, b_2\}$ using a $\top$-connection to D means we connect $\{d_1, d_3, d_5, d_7\}$ to b_1 and $\{d_2, d_4, d_6, d_8\}$ to b_2 or vice versa. (In particular, complementing a connection does not change its type.) In Appendix B, we prove that the gadget satisfies the claimed properties (Lemmas 6 and 7).

Hardness result. Using the blocking gadget, we prove that approximating an optimal solution in the number of vertices of the graph is computationally hard.

▶ **Theorem 5.** *For any $\delta > 0$, unless* P = NP*, there is no polynomial-time algorithm approximating the optimal solution of an instance of Problem 1 to within a factor of $n^{\frac{1}{3}-\delta}$.*

Proof sketch, full proof in Appendix B. We use a reduction from graph coloring. Each vertex in the graph coloring instance becomes a color class of size 2. Merging these classes corresponds to assigning the represented vertices the same color. The blocking gadget represents edges. It prevents color classes representing adjacent vertices in the original graph from getting merged. To obtain the stated hardness bounds, an additional blowup step is necessary. ◀

Given the approximation hardness result, we turn to heuristics to merge colors in practice.

6 Merging Colors in Practice

We now describe our practical approach to compress graphs by merging colors. The practical approach exploits situations in which colors are easily and efficiently mergeable. First, we state more easily checkable consistency conditions, under which colors can be merged. We will find that the hardness result of Section 5 also applies in this setting. Then, we devise an algorithm that greedily searches for easily mergeable color classes.

6.1 Mergeability in Practice

We begin by formalizing what we mean by *easily mergeable*, which will turn out to be a special case of mergeability.

Easy Mergeability. We say color classes C_1 and C_2 are *easily mergeable* if the following three properties hold:

1. There exists an equivalence-ensuring bijection M between C_1 and C_2, satisfying one of the following:
   a. The edges between C_1 and C_2 in G describe a perfect matching in $G[C_1 \cup C_2]$ which induces M. That is, every $v \in C_1$ has exactly one neighbor $M(v) \in C_2$, and vice versa.
   b. There is a sequence of color classes $C_1 = D_1, \dots, D_m = C_2$, such that each (D_i, D_{i+1}) for $i \in [1, m-1]$ satisfies Condition (a). Let $M_i : D_i \to D_{i+1}$ denote the induced bijections. Then $M = M_1 \circ \dots \circ M_m$ must hold.
2. Considering the inner edges of C_1 and C_2, one of the following must hold:
   a. $G[C_2]$ is the empty graph.
   b. The bijection M is an isomorphism between the color classes, i.e., $M(G[C_1]) = G[C_2]$.
3. Every color class $D \notin \{C_1, C_2\}$ satisfies one of the following:
   a. D is adjacent to C_1 *but not* C_2.
   b. For every $v \in C_1$, the vertices v and $M(v)$ are twins with respect to D. That is, for all $d \in D$, $v \sim d \iff M(v) \sim d$ holds.
   c. D is adjacent to C_2 *but not* C_1.

An example for two easily mergeable color classes can be found in Figure 3.

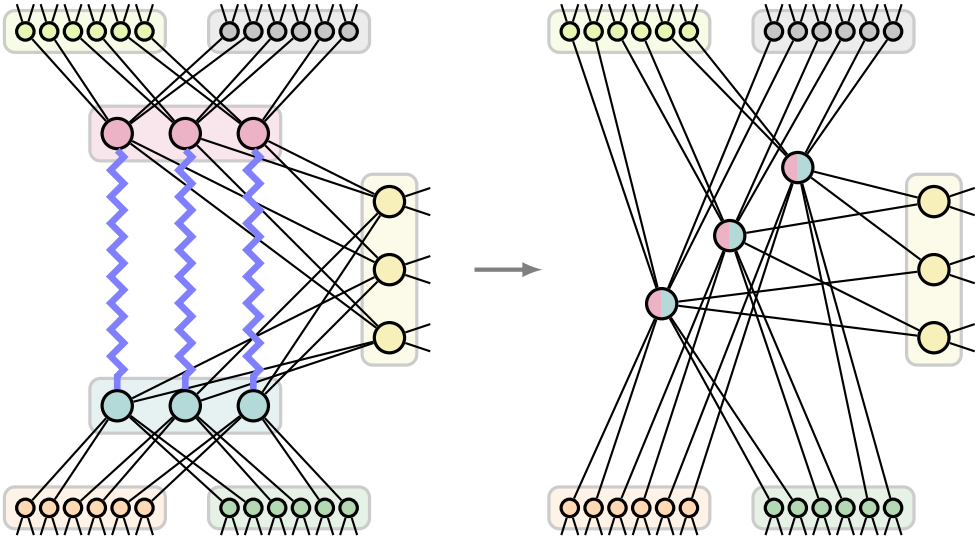

Figure 3 Illustration of the practical, easy mergeability. The matched vertices of the red and blue colors are twins with respect to yellow. The colors of all other connected vertices are disjoint. Hence, the red and blue colors can be merged.

The reduced graph. Let C_1 and C_2 be easily mergeable color classes of G. First observe that 1a immediately ensure that M is an equivalence-ensuring bijection. For 1b this follows by induction on m using the argument for 1a repeatedly. We define the suitable replacement graph G' (with respect to C_1, C_2 and M) as follows.

As before, let $\mathcal{D}$ be the set of color classes that are neighbors of C_2 but not in $\{C_1, C_2\}$ and $V_{\mathcal{D}} \coloneqq \bigcup_{D\in\mathcal{D}} D$ be the vertices in the classes in $\mathcal{D}$. The vertex set of G' is $V_{\mathcal{D}} \cup C_1$. Regarding the edge set, the internal edges of C_1 are contained in G', meaning $G'[C_1] = G[C_1]$. For $D \in \mathcal{D}$, in Cases 3a and 3b, the edges between D and C_1 induce $G[D, C_1]$. In Case 3c, we have $E(G') \cap (D \boxtimes C_1) = \{\{M^{-1}(v), u\} \mid \{v, u\} \in E(G) \cap (C_2 \boxtimes D), v \in C_2\}$.

We briefly argue that G' is a suitable replacement graph: here the crucial requirement that we have to verify is $\mathrm{Aut}(G') = \mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))|_{V_{\mathcal{D}}\cup C_1}$. For $\mathrm{Aut}(G') \subseteq \mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))|_{V_{\mathcal{D}}\cup C_1}$, consider a permutation $\varphi \in \mathrm{Aut}(G')$ and its extension to C_2 using M. We analyze separately whether the edges within C_2 and whether the edges in $C_2 \boxtimes D$ for $D \in \mathcal{D}$ are respected by this extension. For edges inside C_2, we observe that 2a or 2b guarantees compatibility of the internal action on C_2. For edges between C_2 and a neighboring class, we observe that 3a, 3b, or 3c guarantee compatibility of incidences with neighboring classes. Similarly, we can also see that $\mathrm{Aut}(G') \supseteq \mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))|_{V_{\mathcal{D}}\cup C_1}$ holds, since all automorphisms of the original graph reduced to C_1, C_2 and the neighboring colors of C_2 respect M and the edges of G'.

This gives rise to a reduced graph $G_{C_1 \approx_M C_2}$ as described in Section 4. Thus easy mergeability is a special case of mergeability.

Hardness. The hardness result of Theorem 5 also holds for easy mergeability, since all merges used in Theorem 5 satisfy easy mergeability: all mergeable colors are connected by a direct matching, there are no internal edges in any of the colors, and the connections to other colors are either disjoint or twinned. Since easy mergeability is a special case of mergeability, all blocked merges are also blocked w.r.t. easy mergeability.

6.2 Practical Algorithm

We now describe an algorithm to check easy mergeability. The algorithm starts from a candidate color C, and attempts to merge many other color classes with C. It does so by performing a special breadth-first search, that traverses colors along direct one-to-one matchings between the colors. While performing this traversal, the algorithm maintains data structures enabling it to check consistency with everything merged with C so far.

Description of Algorithm 1. Algorithm 1 formalizes the aforementioned traversal. The algorithm maintains a queue of color classes to be explored, starting from the initial color C. During the traversal, it builds a mapping M that identifies vertices of newly discovered colors with vertices of C, by composing one-to-one matchings encountered along the search.

The algorithm attempts to merge newly discovered color classes one-by-one with C. While doing so the algorithm maintains data structures that represent the neighborhood of C together with the color classes L that were already successfully merged.

In particular, for every class in $L \cup \{C\}$, the algorithm maintains the neighbors for each neighboring color c separately, namely in the set $G_C[c]$. When a new color class D is reached, the algorithm analogously constructs, for all neighboring colors c of D, edge sets $G_D[c]$, containing the connections from D to c. (More precisely, these connections are already translated from D to C via M. Intuitively, we store the edges that would occur if we were to merge D with C via M.)

Organizing the neighbors by color allows us to check edge consistency for each neighboring color separately. In particular, for easy mergeability, we only need to check colors that are adjacent to *both* D and C together with the already-merged color classes L. Hence, it suffices to check the neighboring colors of D.

To check consistency, G_D is compared with the accumulated structure G_C, ensuring that all adjacencies agree and that the internal structure of D is compatible under M. A special case arises from connections from D to C or any class in L. Here, the algorithm requires that the connections form a one-to-one matching that is compatible with M. If all checks succeed, D is added to the list of already-merged colors L, and G_C is updated to incorporate new edges stemming from D. Edges from D to C or L are discarded. Having finished D, the temporary data structures for the neighborhood of D are reset, and the algorithm considers the next color class in the queue.

Runtime of Algorithm 1. We argue that Algorithm 1 can be implemented to run in $\mathcal{O}(|V| + |E|)$. Clearly, everything up to the main loop over the queue S runs in time $\mathcal{O}(|V| + |E|)$.

Each color class is enqueued at most once due to marking, and thus the main loop processes each color class D only once. It therefore suffices to show that every iteration corresponding to a color class D runs in time $\mathcal{O}(|N(D)|)$.

Constructing G_D requires traversing all edges incident to D, which takes $\mathcal{O}(|N(D)|)$ time. To perform the consistency check efficiently, we can maintain alongside G_D a list of colors c for which $G_D[c]$ is non-empty. This list can be built during the traversal of $N(D)$. The check then only inspects these colors, and hence also runs in $\mathcal{O}(|N(D)|)$ time. The same list also allows updating G_C and resetting G_D within the same bound.

Implementation of Algorithm 1. In practice, we begin by selecting an arbitrary color class C and run Algorithm 1. We collect all merges that can be applied. Then, we select a new color class C' that has not yet been merged. More precisely, we now only consider color classes that have not been marked in previous calls to Algorithm 1, and whose neighbors are not involved in any previously recorded merge.

Algorithm 1 The algorithm used to find merges for a given color.

```
function find-merge
    Input : ➢ Graph G = (V, E, π)
            ➢ Color class C
    Output : ≺ Set of color classes L that can be merged with C
initialize empty set L;
initialize arrays of sets G_C, G_D of size |V(G)|;
initialize map of vertices M of size |V(G)|;
set M[v] := v for all v ∈ C;
// collect outgoing edges of C per color
for u ∈ C do for v ∈ N(u) do add (u, v) to G_C[π(v)] ;
initialize empty queue S;
enqueue C to S;
mark C;
while S not empty do
    D := dequeue from S;
    // enqueue neighboring color classes matched to D
    for each neighboring color class D' of D do
        if D' is marked then continue;
        if D and D' are connected by matching M' : D' → D then
            M[v] := M[M'(v)] for all v ∈ D';
            enqueue D' to S;
            mark D';
    if C == D then continue;
    // collect outgoing edges of D per color, translated to C
    for u ∈ D do for v ∈ N(u) do add (M[u], v) to G_D[π(v)] ;
    // check whether D is consistent with C merged with L
    check := true;
    for all i where G_D[i] is non-empty do
        B := π^{-1}(i);
        if B ∈ L ∪ {C} then
            check := (check and ({(M[l], l) | l ∈ B}== G_D[i]));
            continue;
        if G_C[i] is empty then continue;
        check := (check and (G_C[i] == G_D[i]));
    check := (check and (G[C] == M(G[D]) or G[D] is empty));
    // update L and C neighborhood
    if check then
        L := L ∪ {D};
        for all i where G_D[i] is non-empty do
            if π^{-1}(i) ∉ L ∪ {C} then G_C[i] := G_D[i] ;
    reset all sets in G_D to the empty set;
return L
```

Table 1 Summary of experimental results per dataset: ρ and *Red. to* indicate the ratio between total final size without and with MERGE, T^{rule} is the runtime of the MERGE rule, T^{org} and T^{new} are the total runtimes of DEJAVU without and with the MERGE rule, and $T^{\text{org}}/T^{\text{new}}$ is the speedup. Data reduction considers fewer instances as those solved during preprocessing (reduced to size 0) are excluded.

	Data Reduction (Section 7.1)				Runtime (Section 7.2)					
Set	# Inst.	Size	ρ	Red. to	# Inst.	Size	T^{rule}	T^{org}	T^{new}	Speedup
COMB	2421	2.3 GB	1.02	98.04%	3160	4.6 GB	1 s	2365 s	2357 s	1.00
MIPLIB	369	6.4 GB	5.32	18.80%	1026	13.4 GB	13 s	307 s	224 s	1.37
PACE23	161	2.2 GB	2.29	43.67%	400	3.5 GB	2 s	44 s	47 s	0.94
SAT23	242	122.0 GB	1.48	67.57%	397	131.6 GB	10 s	1300 s	1292 s	1.01
SAT24	184	99.1 GB	48.98	2.04%	400	110.8 GB	327 s	4402 s	2147 s	2.05
SAT25	169	82.9 GB	6.01	16.64%	387	133.0 GB	136 s	2736 s	1949 s	1.40
Total	3546	315.0 GB	8.56	11.68%	5770	396.8 GB	488 s	11154 s	8016 s	1.39
↳ w/o COMB	1125	312.7 GB	11.40	8.77%	2610	392.3 GB	487 s	8789 s	5659 s	1.55

We repeat this process until no further such color classes remain. At that point, we update the graph data structure to reflect all collected merges and iterate the procedure on the modified graph up to three times.

7 Experimental Study

In this section, we evaluate the effectiveness of Algorithm 1 and its impact on the overall runtime of symmetry detection. The tool DEJAVU already incorporates the graph simplification techniques as described in [4, 1]. We assess our new approach *in addition* to these existing preprocessing techniques. Specifically, we implemented our algorithm as an additional preprocessing step in DEJAVU, written in C++. In the following, we refer to this extended version as DEJAVU with the MERGE rule. While randomized, DEJAVU is currently the fastest tool for symmetry detection across a wide range of graphs (see [1] for a recent comparison).

All experiments are carried out on machines with two Intel Xeon Gold 6130 CPUs (2.10 GHz, 16 cores each) and 384 GB RAM. Despite DEJAVU being a sequential tool, we grant each run exclusive access to a machine to avoid interference from other processes, and to ensure that the results are not affected by resource contention. We repeat each measurement 5 times and report their median; the variance is negligible for all discussions.

We evaluate the performance of the MERGE rule on a large collection of graph classes, including various standard datasets. In total there are 5770 instances with a file size of 396.8 GB in the DIMACS format. As detailed in Table 1, they are categorized as follows:

- SAT23, SAT24, and SAT25 are based on the SAT Competition[2] datasets of the respective years. They are converted to the DIMACS format using the SATSUMA [2] tool. Two SAT23 instances are excluded for exceeding the time limit of 20 min.
- COMB is a standard benchmark library for testing graph isomorphism algorithms[3]. It contains a wide variety of graphs. Many of them are highly-regular graphs and graphs designed to be challenging for graph isomorphism solvers. Most of these are small in size compared to the graphs tested in other sets.

[2] `https://www.satcompetition.org`

[3] `https://pallini.di.uniroma1.it/Graphs.html`: all of BLISS distribution without `cfi` and `k` (see below), CONAUTO distribution, SAUCY distribution, random graphs, miscellaneous graphs, Dawar-Yeung graphs `https://automorphisms.org/`: `cfixl`, `groups128`, `kxl`.

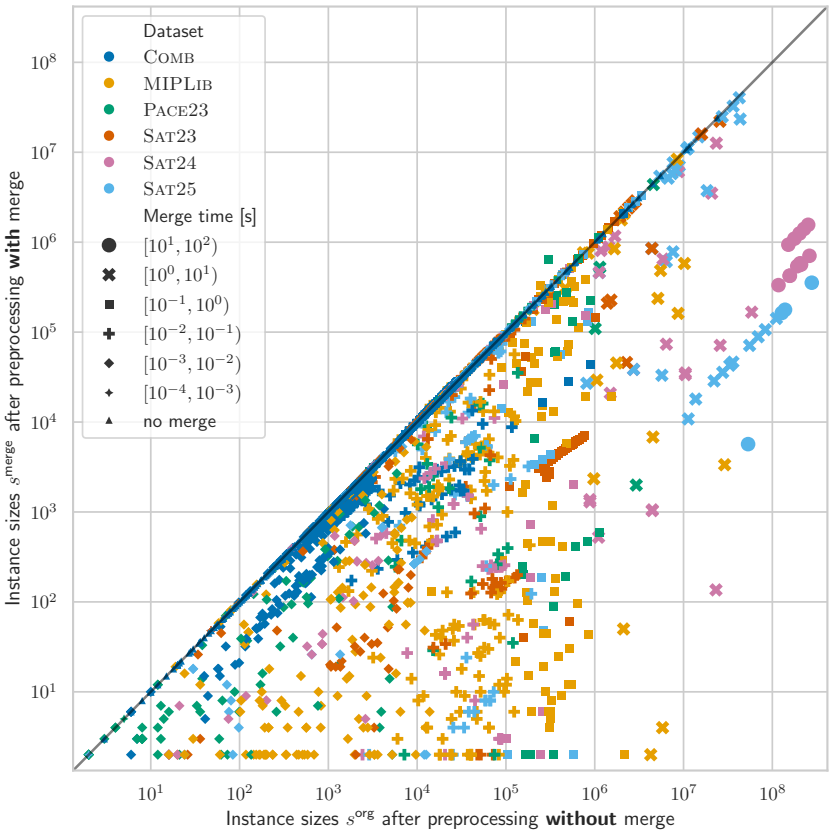

Figure 4 Instance sizes s_i^{org} and s_i^{new} after preprocessing with and without the merging rule. The diagonal line indicates $\rho_i = 1$, where the rule does not change the graph size; lower is better.

- PACE23 contains instances of the PACE Challenge 2023 [5] on twin-width. Many of the graphs contain twins and very sparse symmetries.
- MipLib contains instances stemming from symmetry exploitation in mixed-integer solvers, in particular from instances of the MIPLIB 2017 [13]. The original instances were translated into graphs using SCIP [14].

For both SATSUMA and SCIP, instances are encoded as graphs after presolving, which is the default procedure. Consequently, any instances that are solved prior to computing symmetries are excluded.

7.1 Data reduction

The DEJAVU tool already implements a number of preprocessing steps, capable of solving 2224 easy[4] instances directly, most of which are asymmetric. As no further data compression is possible, we exclude these instances from the data reduction analysis in this section. A summary for each of the sets can be found in Table 1.

Let s_i^{org} and s_i^{new} be the graph size (number of nodes and edges) of instance i after preprocessing without and with the Merge rule, respectively. Then, we define *data reduction* $\rho_i = s_i^{\text{org}}/s_i^{\text{new}}$ as the ratio between both; a value of $\rho_i > 1$ indicates that the Merge rule has reduced the graph size, while $\rho_i < 1$ indicates a regression.

[4] Despite accounting for 38% of instances, only 5.30% of the total runtime is spent on these instances.

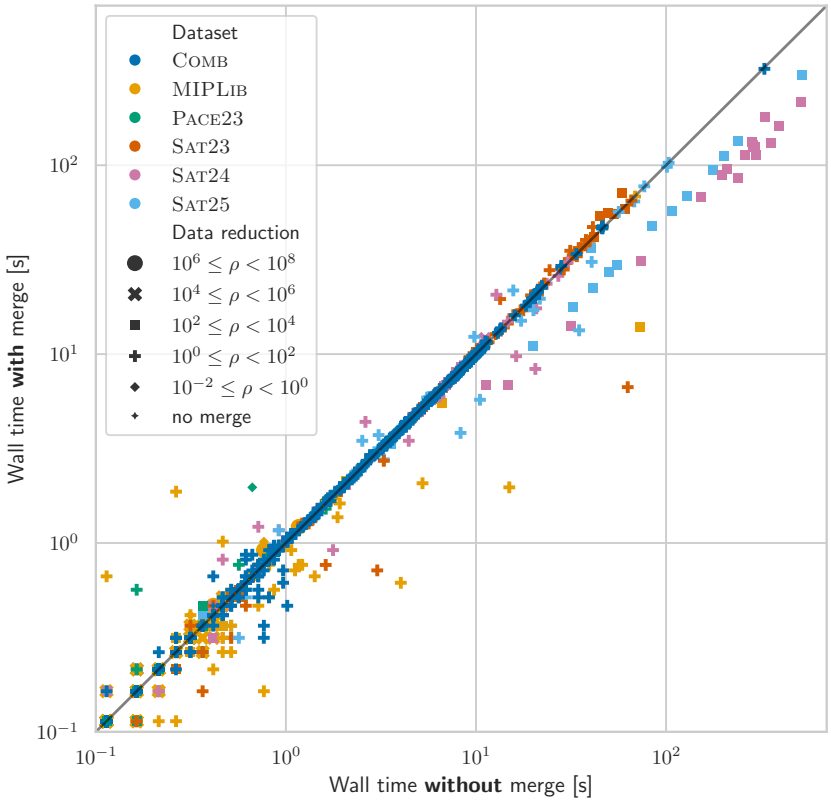

Figure 5 Runtime of DEJAVU with and without the merging rule, respectively. The diagonal line indicates no change; lower is better.

Aggregated by total size, we achieve a data reduction of 8.56 across all instances, which corresponds to a remaining size of 11.7% compared to preprocessing without the MERGE rule. On the practical sets (all w/o COMB), the technique applies to the large majority of instances, in particular to 81.8% (920 of 1125). Data reduction is meaningful across many instance sizes and types, as indicated in Figure 4. On COMB, reductions are very limited – as is expected for highly regular and challenging graphs. Surprisingly, however, our new technique still yields reductions on some families of combinatorial instances (e.g., Miyazaki graphs, grid graphs, and graphs stemming from groups). This is in contrast to prior preprocessing techniques, which typically show no effect on these families [4].

The observed gains are driven almost entirely by the rule itself; effects on subsequent preprocessing rules are negligible. Only 6 instances exhibit an increase in graph size, due to an interaction with the HUB rule (see Section 5.5.2 in [1]).

7.2 Runtime

To obtain a realistic performance characterization, we consider all 5770 instances when measuring runtime (including those solved during preprocessing). We find that the MERGE rule leads to a speedup of 1.39, reducing the total runtime from 11153s to 8016s. The overhead is almost always negligible, as indicated in Figure 5. More precisely, executing the new algorithm accounts for 6.1% of the total runtime (488.1s out of 8016s).

While the compression applies across different instance sizes, in terms of runtime it seems particularly effective on large instances that take long in the unmodified algorithm. If we consider the 32 instances on which the unmodified DEJAVU requires more than 60s, we find a speedup of 1.87– only one instance exhibits an (insignificant) slowdown.

To assess the robustness of our methods, we additionally tested *shuffled* versions of the above datasets, where we apply a random permutation to the node and color indices. While this variant exhibits significantly higher absolute runtimes (19467s without MERGE rule and 13765s with the rule), the relative speedup of 1.41 is qualitatively similar.

8 Conclusions and Future Work

We studied compressing graphs while preserving their symmetries, by merging color classes under local conditions. We showed that an optimal sequence of these merges is hard to obtain. Nevertheless, we gave an efficient algorithm that achieves substantial compression ratios on practical inputs. Furthermore, this resulted in a significant speedup for the symmetry-detection algorithm DEJAVU.

Interesting future work remains: subsequently applied symmetry handling algorithms could be adapted to work on the compressed structure directly. Furthermore, it would be interesting to generalize the practical algorithm to incorporate more cases. Lastly, there is the more theoretical matter of computing a graph that expresses the intersection of the automorphism groups of two given graphs, or deciding that it does not exist. Determining the computational complexity of this problem seems both intriguing and challenging.

References

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

2 Markus Anders, Sofia Brenner, and Gaurav Rattan. Satsuma: Structure-based symmetry breaking in SAT. *J. Artif. Intell. Res.*, 85, 2026. doi:10.1613/JAIR.1.18744.

3 Markus Anders and Pascal Schweitzer. Parallel computation of combinatorial symmetries. In *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.

4 Markus Anders, Pascal Schweitzer, and Julian Stieß. Engineering a preprocessor for symmetry detection. In Loukas Georgiadis, editor, *21st International Symposium on Experimental Algorithms, SEA 2023, Barcelona, Spain, July 24-26, 2023*, volume 265 of *LIPIcs*, pages 1:1–1:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SEA.2023.1.

5 Max Bannach and Sebastian Berndt. PACE Solver Description: The PACE 2023 Parameterized Algorithms and Computational Experiments Challenge: Twinwidth. In Neeldhara Misra and Magnus Wahlström, editors, *18th International Symposium on Parameterized and Exact Computation (IPEC 2023)*, volume 285 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:14, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.IPEC.2023.35.

6 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*, pages 148–159. Morgan Kaufmann, 1996.

7 Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In *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.

8 Paul T. Darga, Karem A. Sakallah, and Igor L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 149–154. ACM, 2008. doi:10.1145/1391469.1391509.

9 David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *International Conference on Automated Deduction*, pages 222–236. Springer, 2011. doi:10.1007/978-3-642-22438-6_18.

10 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 83–100. Springer, 2017. doi:10.1007/978-3-319-66263-3_6.

11 Ian P. Gent, Tom W. Kelsey, Steve Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In Peter van Beek, editor, *Principles and Practice of Constraint Programming – CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2005. doi:10.1007/11564751_21.

12 Ian P Gent, Karen E Petrie, and Jean-François Puget. Symmetry in constraint programming. *Foundations of Artificial Intelligence*, 2:329–376, 2006. doi:10.1016/S1574-6526(06)80014-3.

13 Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. *Mathematical Programming Computation*, 2021. doi:10.1007/s12532-020-00194-3.

14 Christopher Hojny, Mathieu Besançon, Ksenia Bestuzheva, Sander Borst, João Dionísio, Johannes Ehls, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Adrian Göß, Alexander Hoen, Jacob von Holly-Ponientzietz, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Marco Lübbecke, Stephen J. Maher, Paul Matti Meinhold, Gioni Mexi, Til Mohr, Erik Mühmer, Krunal Kishor Patel, Marc E. Pfetsch, Sebastian Pokutta, Chantal Reinartz Groba, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Matthias Walter, Dieter Weninger, and Liding Xu. The SCIP optimization suite 10.0, 2025. arXiv:2511.18580.

15 Annika Jäger and Marc E. Pfetsch. Structure-preserving symmetry presolving for mixed-binary linear problems. Technical report, Optimization Online, 2025. Technical report. URL: https://optimization-online.org/2025/11/structure-preserving-symmetry-presolving-for-mixed-binary-linear-problems/.

16 Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007. doi:10.1137/1.9781611972870.13.

17 Tommi A. Junttila and Petteri Kaski. Conflict propagation and component recursion for canonical labeling. In *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.

18 Volker Kaibel, Matthias Peinhardt, and Marc E. Pfetsch. Orbitopal fixing. *Discrete Optimization*, 8(4):595–610, 2011. doi:10.1016/j.disopt.2011.07.001.

19 François Margot. Pruning by isomorphism in branch-and-cut. *Math. Program.*, 94(1):71–90, 2002. doi:10.1007/S10107-002-0358-2.

20 Brendan D. McKay. Practical graph isomorphism. In *10th. Manitoba Conference on Numerical Mathematics and Computing (Winnipeg, 1980)*, pages 45–87, 1981.

21 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.

22 Gioni Mexi, Dominik Kamp, Yuji Shinano, Shanwen Pu, Alexander Hoen, Ksenia Bestuzheva, Christopher Hojny, Matthias Walter, Marc E. Pfetsch, Sebastian Pokutta, and Thorsten Koch. State-of-the-art methods for pseudo-boolean solving with SCIP, 2025. arXiv:2501.03390.

23 James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Orbital branching. *Mathematical Programming*, 126(1):147–178, 2011. doi:10.1007/S10107-009-0273-X.

24 R. Parris and R. C. Read. Graph isomorphism and the coding of graphs. Technical Report UWI/CC5, University of the West Indies, Jamaica, 1967.

25 Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008. arXiv:0804.4881.

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

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

28 Alasdair Urquhart. The symmetry rule in propositional logic. *Discret. Appl. Math.*, 96-97:177–193, 1999. doi:10.1016/S0166-218X(99)00039-6.

29 David Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 681–690, 2006. doi:10.1145/1132516.1132612.

A Proof of Correctness for Mergeability

We give the missing proof that mergeability suffices to preserve automorphisms in the reduced graph.

▶ **Theorem 4.** *Let G be a graph, M an equivalence-ensuring bijection for color classes C_1, C_2, and G' a suitable replacement graph. Then* $\mathrm{Aut}(G) = \mathrm{Aut}(G_{C_1 \approx_M C_2}) \uparrow_M$.

Proof. We show the two directions.

- *(Inclusion $\mathrm{Aut}(G) \subseteq \mathrm{Aut}(G_{C_1 \approx_M C_2}) \uparrow_M$):* Suppose $\varphi \in \mathrm{Aut}(G)$. Then $\varphi|_{V_{\mathcal{D}} \cup C_1 \cup C_2}$ is an automorphism of $G[V_{\mathcal{D}} \cup C_1 \cup C_2]$. Moreover, $\varphi|_{V_{\mathcal{D}} \cup C_1 \cup C_2}$ respects M since M is equivalence-ensuring and thus respected by $\mathrm{Aut}(G)$. It follows that $\varphi' \coloneqq \varphi|_{V_{\mathcal{D}} \cup C_1} \in \mathrm{Aut}(G')$, since G' is a suitable replacement graph.
  It suffices now to argue that $\widetilde{\varphi} \coloneqq \varphi|_{V(G) \setminus C_2}$ is in $\mathrm{Aut}(G_{C_1 \approx_M C_2})$ because φ is a lift of $\widetilde{\varphi}$ that respects M. To see that $\widetilde{\varphi} \in \mathrm{Aut}(G_{C_1 \approx_M C_2})$ we consider a pair of possibly equal color classes B_1, B_2 of the reduced graph (so they are not C_2) and distinguish two cases:
  - *(parts that were modified)* If $B_1 = C_1$ and $B_2 \in \{C_1\} \cup \mathcal{D}$ then $\widetilde{\varphi}$ preserves adjacency and non-adjacency in $E(G_{C_1 \approx_M C_2}) \cap B_1 \boxtimes B_2$ because $\varphi' \in \mathrm{Aut}(G')$.
  - *(parts that were not modified)* If $B_1 \neq C_2$ and $B_2 \notin \{C_1, C_2\} \cup \mathcal{D}$ then $\widetilde{\varphi}$ respects adjacency and non-adjacency in $E(G_{C_1 \approx_M C_2}) \cap B_1 \boxtimes B_2$ since $E(G_{C_1 \approx_M C_2}) \cap B_1 \boxtimes B_2 = E(G) \cap B_1 \boxtimes B_2$.

  Overall $\widetilde{\varphi} \in \mathrm{Aut}(G_{C_1 \approx_M C_2})$.
- *(Inclusion $\mathrm{Aut}(G) \supseteq \mathrm{Aut}(G_{C_1 \approx_M C_2}) \uparrow_M$):* Next, suppose $\varphi \in \mathrm{Aut}(G_{C_1 \approx_M C_2}) \uparrow_M$. We argue that $\varphi \in \mathrm{Aut}(G)$.
  We argue similarly to the previous case. For edges not involving C_2 the argument is exactly the same as for the other direction.
  Assume now that $B_1 = C_2$ or $B_2 = C_2$. In that case we may assume that $B_1 = C_2$ and thus that $B_2 \in \mathcal{D} \cup \{C_1, C_2\}$. Note that $\varphi|_{V_{\mathcal{D}} \cup C_1} \in \mathrm{Aut}(G')$. Thus $\varphi|_{V_{\mathcal{D}} \cup C_1} \in \mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))|_{V_{\mathcal{D}} \cup C_1}$ since G' is a suitable replacement graph. (Here $\widetilde{G}$ is again the graph obtained from $G[V_{\mathcal{D}} \cup C_1 \cup C_2]$ by deleting all edges that do not have at least one

endpoint in C_1 or C_2.) This means that there is an automorphism in $\mathrm{Stab}_M(\mathrm{Aut}(\widetilde{G}))$ of which $\varphi|_{V_\mathcal{D} \cup C_1}$ is a restriction. Since φ respects M and lifts are unique, φ must be that automorphism. We conclude that $\varphi|_{V_\mathcal{D} \cup C_1 \cup C_2} \in \mathrm{Aut}(\widetilde{G})$. It follows that φ also respects adjacency and non-adjacency for edges with one endpoint in C_2. Thus φ is an automorphism of G. ◀

B Proofs for Hardness of Optimal Reduction

We give the missing proofs for proving the approximation hardness result for optimal merging.

Let us first argue that the blocking gadget satisfies the claimed properties. Crucially, in the following, we assume that the vertices that lie in the same color class of size 2 are in the same orbit. This will be the case in our constructions below.

▶ **Lemma 6.** *Let $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ be two color classes in G, such that $M(a_i) = b_i$ is an equivalence-ensuring bijection and there exists an automorphism $\varphi \in \mathrm{Aut}(G)$ with $\varphi(a_1) = a_2$. Furthermore, let D be a blocking gadget in G, such that A is connected to D using a $\bot$-connection, and B using a $\top$-connection. Then, there is no suitable replacement graph for merging A and B.*

Proof. Let us apply the automorphism φ to D. In particular, from the existing $\bot$-connection and $\top$-connection we can conclude that φ exchanges the set $\{d_1, d_3\}$ with $\{d_6, d_8\}$ and the set $\{d_2, d_4\}$ with $\{d_5, d_7\}$. Even when the vertices of A and B are fixed, vertices inside the remaining sets of size 2 are still freely interchangeable in D. (Note that this holds no matter whether the $\bot/\top$ connections are complemented or not.)

Any suitable replacement graph merging A and B requires us to express this local automorphism group over the vertices of D, using the connections from a single color class $C = \{c_1, c_2\}$. This means, for each set, we need to either connect all their contained vertices to c_1 (or c_2), or none of them. Let us say without loss of generality we connect $\{d_1, d_3\}$ to c_1. In order to enforce the exchange of sets, we are now forced to connect $\{d_6, d_8\}$ to c_2. For $\{d_2, d_4\}$, we can now decide whether to attach this set to c_1, c_2, both, or none. In any case, it is now easy to see that this fails to encode the correct automorphism group:

- If $\{d_2, d_4\}$ is connected to c_1, this entails that $\{d_1, d_2, d_3, d_4\}$ are interchangeable in D.
- If $\{d_2, d_4\}$ is connected to c_2, this entails that $\{d_2, d_4, d_6, d_8\}$ are interchangeable in D.
- If $\{d_2, d_4\}$ is connected to both or none of c_1, c_2, we are not enforcing the exchange of $\{d_2, d_4\}$ with $\{d_5, d_7\}$ when c_1 and c_2 are flipped.

In particular, none of these match the required local automorphism group. ◀

Furthermore, we observe that the $\top$ and $\bot$ connections can only be rewritten to their respective complements, while preserving local automorphisms.

▶ **Lemma 7.** *Let $A = \{a_1, a_2\}$ and $B = \{b_1, b_2\}$ be two color classes in G, such that $M(a_i) = b_i$ is an equivalence-ensuring bijection and there exists an automorphism $\varphi \in \mathrm{Aut}(G)$ with $\varphi(a_1) = a_2$. Furthermore, let D be a blocking gadget in G, such that A is connected to D using a $\bot$-connection. Assume B is either not connected to D, or connected to D using a $\bot$-connection. Assume there exists a suitable replacement graph G' for the merge of A and B. Then, $G'[A \cup D]$ is a $\bot$-connection.*

Proof. Any replacement graph needs to retain the local automorphisms on D. We observe that the $\bot$-connection from A to D imposes the following restrictions: fixing A leaves the vertices in $D_{1,4} = \{d_1, ..., d_4\}$ and $D_{5,8} = \{d_5, ..., d_8\}$ freely interchangeable, whereas applying φ exchanges $D_{1,4}$ with $D_{5,8}$. (If B is also connected to D, this imposes the same restriction.)

This means, for each of the two blocks, either all of them must connect to a_1 (or a_2) in G', or none of them. Exchanging a_1 and a_2 needs to ensure that $D_{1,4}$ and $D_{5,8}$ are exchanged in D. This implies that one of them has to be connected to a_1, and the other to a_2. There are two ways to accomplish this: the $\bot$-connection, or its complement. ◀

The analogous result of Lemma 7 holds for the $\top$-connection. We now give the main hardness result.

▶ **Theorem 5.** *For any $\delta > 0$, unless $\mathsf{P} = \mathsf{NP}$, there is no polynomial-time algorithm approximating the optimal solution of an instance of Problem 1 to within a factor of $n^{\frac{1}{3}-\delta}$.*

Proof. We use a reduction from graph coloring. Given an uncolored graph G, we are supposed to find the least k, such that there is a coloring $\pi\colon V(G) \to [1,k]$ which for each $\{u,v\} \in E(G)$ ensures that $\pi(u) \neq \pi(v)$ holds. Graph coloring is NP-hard. Furthermore, unless $\mathsf{P} = \mathsf{NP}$, for every $\epsilon > 0$, there is no polynomial-time algorithm approximating the optimal number of colors k to within a factor of $n^{1-\epsilon}$ [29].

Construction. Given an instance of graph coloring G, we transform G into an instance of Problem 1, i.e., a colored graph G'. Let $t \in \mathbb{N}$ be a constant which we determine later. We call each $i \in [1,t]$ a *layer* of our construction.

- For each $v \in V(G)$ we create for every $i \in [1,t]$ a color class of size 2 containing $a_{v,i}$ and $b_{v,i}$ in G'. We set $C_{v,i} \coloneqq \{a_{v,i}, b_{v,i}\} \subseteq V(G')$.
- For every pair of distinct vertices $u, v \in V(G)$ we connect $a_{u,i}$ with $a_{v,i}$ and $b_{u,i}$ with $b_{v,i}$ in G', for every $i \in [1,t]$.
- For each $e = \{u,v\} \in E(G)$, we add a blocking gadget D_e to the graph. For every $i \in [1,t]$ we connect $C_{u,i}$ using a $\bot$-connection to D_e, and $C_{v,i}$ using a $\top$-connection.
- Let $\lceil \log(t) \rceil = k$. We add blocking gadgets E_j for $j \in [1,k]$. For every $j \in [1,k]$, $i \in [1,t]$, and $v \in V(G)$, we connect $C_{v,i}$ to E_j using a $\top$-connection if the j-th bit of the binary representation of i is 1, and using a $\bot$-connection otherwise.

Equivalent Colors. First, let us observe which color classes can or cannot be merged in the initial graph G'. For all $u, v \in V(G)$ and $i \in [1,t]$ where $\{u,v\} \notin E(G)$, we can merge $C_{v,i}$ and $C_{u,i}$. First, they are connected by a matching. This matching is equivalence-ensuring, so it is obvious that they indeed consist of equivalent orbits. No blocking gadget D_e was placed between them, and since they are on the same layer i, their connections to all E_j gadgets are either both $\top$ or both $\bot$ connections. Hence, they can be merged.

If there exists an edge $\{u,v\} \in E(G)$, then there is a blocking gadget $D_{\{u,v\}}$ between $C_{u,i}$ and $C_{v,i}$, meaning from Lemma 6 it follows that u and v cannot be merged.

Whenever for two classes $C_{u,i}$ and $C_{v,j}$ it holds that $i \neq j$, in the binary representation of i and j there must be a position k in which they differ. Consequently, the connection types to the blocking gadget E_k differs, meaning from Lemma 6 it follows that the classes cannot be merged.

None of the blocking gadgets can be merged, since for each blocking gadget there is a non-trivial automorphism that fixes all vertices outside the gadget. Thus no blocking gadget is equivalent to any other color class.

Blocked merges prevail. Consider the merge $A = C_{u,i}$ and $B = C_{v,i}$. Let X_A denote all color classes of size 2 with which A can *not* be merged before merging with B (note that $B \notin X_A$). Analogously, let X_B denote all color classes of size 2 with which B can *not* be merged before merging with A (note that $A \notin X_B$).

From Lemma 6 it follows that A and B must have the same connection type to all blocking gadgets. In turn, from Lemma 7, it follows that in the suitable replacement graph for A and B, all connection types to blocking gadgets are retained. After merging, the resulting color class is thus blocked from being merged with the colors $X_A \cup X_B$. Observe that the argument inductively holds for colors A and B that result from previous merges: if A resulted from merging colors $\{A_1, \ldots, A_k\}$ in G', A can be merged with B representing $\{B_1, \ldots, B_l\}$ whenever $\{B_1, \ldots, B_l\} \cap (X_{A_1} \cup \ldots \cup X_{A_k}) = \emptyset$. We conclude that in G', merges commute and the order is of no importance.

Let us observe what merging colors means in our construction. When merging colors $C_{u,i}$ and $C_{v,i}$, the resulting color can only continue to be merged with vertices that are *neither* adjacent to u, *nor* to v in G. In turn, if the classes have already been merged and represent sets of vertices U and V, we can merge these classes whenever $N(U) \cap N(V) = \emptyset$ holds. Hence, more generally, we can merge the colors representing an *independent set* in G into a single color class in G'.

Recovering graph coloring. To summarize, for every layer $i \in [1, t]$ we can precisely merge sets of colors into a single color, that represent independent sets in G. In particular, each remaining color $C_{u,i}$ represents an independent set in G. This exactly corresponds to valid colorings of G. More formally, if k color classes $C_{u,i}$ remain at layer i, this implies we can color G using k colors. On the other hand, if we can color G using k colors, we can shrink layer i to k classes $C_{u,i}$.

However, G' consists of t layers, and includes additional blocking gadgets. Summing over the layers, we conclude that if there is a solution of size s for G', we can color G with at most

$$\left\lceil \frac{s - 8m - 8\lceil \log t \rceil}{2t} \right\rceil$$

colors. (The above essentially takes the average over all layers, which guarantees that at least one of the coloring problems on one of the layers is solved sufficiently well.) On the other hand, if G can be colored with k colors, then we can shrink G' to

$$2kt + 8m + 8\lceil \log t \rceil$$

vertices.

Approximation hardness. Assume for some fixed $\delta > 0$ that we can approximate the optimal solution to Problem 1 to within a factor of $n^{\frac{1}{3}-\delta}$. Let G be a graph given as input to the graph coloring problem. We use the approximation algorithm for Problem 1 to approximate the chromatic number of G.

Let $n = |V(G)|$, $m = |E(G)|$, and k be the chromatic number of G. We choose $t = n^2$ and construct the input G' to Problem 1 using our construction above. The graph G' has $2n^3 + 8m + 16\lceil \log n \rceil$ vertices. Any solution to G' with value s implies that G can be colored with at most

$$\left\lceil \frac{s - 8m - 16\lceil \log n \rceil}{2n^2} \right\rceil$$

colors. The optimal solution of G' is

$$2kt + 8m + 16\lceil \log n \rceil.$$

Our approximation algorithm produces a solution s with

$$s \leq (2n^3 + 8m + 16\lceil \log n \rceil)^{\frac{1}{3}-\delta}(2kn^2 + 8m + 16\lceil \log n \rceil).$$

This implies we can color G with at most

$$\left\lceil \frac{(2n^3 + 8m + 16\lceil \log n \rceil)^{\frac{1}{3}-\delta}(2kn^2 + 8m + 16\lceil \log n \rceil) - 8m - 16\lceil \log n \rceil}{2n^2} \right\rceil$$

colors. We estimate to at most

$$\left\lceil \frac{26^{(\frac{1}{3}-\delta)}n^{3(\frac{1}{3}-\delta)}26kn^2)}{2n^2} \right\rceil < 26^{(\frac{1}{3}-\delta)}13n^{3(\frac{1}{3}-\delta)}k + 1 = \mathcal{O}(kn^{1-3\delta})$$

colors. This suffices to give an $n^{1-\epsilon}$ approximation algorithm for graph coloring. ◀