TECHNISCHE
UNIVERSITÄT
DARMSTADT

Efficient Algorithms for Symmetry Detection

Vom Fachbereich Mathematik
der Technischen Universität Darmstadt
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)
genehmigte

Dissertation

von

Markus Anders

aus Kaiserslautern

Erstgutachter:	Prof. Dr. Pascal Schweitzer
Zweitgutachter:	Prof. Dr. Adolfo Piperno
Drittgutachter:	Prof. Dr. Brendan McKay

Darmstadt, 2024

Efficient Algorithms for Symmetry Detection
Doctoral thesis by Markus Anders
Darmstadt, Technische Universität Darmstadt, 2024

1st Referee: Prof. Dr. Pascal Schweitzer
2nd Referee: Prof. Dr. Adolfo Piperno
3rd Referee: Prof. Dr. Brendan McKay

Tag der Einreichung: 16.05.2024
Tag der mündlichen Prüfung: 17.09.2024

Published by TUprints in 2024.
Veröffentlicht durch TUprints im Jahr 2024.
URN: urn:nbn:de:tuda-tuprints-282571
URL: http://tuprints.ulb.tu-darmstadt.de/28257

Abstract

The use of symmetry dramatically impacts the efficiency of algorithms in various application areas. However, no matter how symmetries are used, there needs to be a way to obtain them efficiently. In practice, symmetries of combinatorial structures are usually computed by modeling said structure as a graph. The automorphisms of this graph then precisely model the symmetries of the combinatorial structure. A so-called practical graph isomorphism solver is then used to compute the automorphism group of the constructed graph. Practical graph isomorphism solvers have been developed for more than half a century. While all state-of-the-art solvers are based on the same principles, namely the so-called individualization-refinement paradigm, there are vast differences among them. The differences in the algorithms, in turn, lead to vast performance differences in practice. While practical graph isomorphism has been considered "solved" multiple times over the past few decades, no single algorithm can efficiently solve all the graphs stemming from the various application areas. Indeed, there are even some particular applications for which state-of-the-art implementations struggle to solve all the graphs efficiently. Ideally, one would like to have a singular algorithm that is fast on all graphs.

In this thesis, we describe the design of a new practical graph isomorphism solver called DEJAVU. Instead of trying to develop a better implementation ad hoc, we perform a theoretical analysis of essential algorithmic ingredients used in state-of-the-art algorithms. A central result is a theoretical model for the backtracking behavior of all state-of-the-art algorithms. Within this model, we prove that a Monte Carlo strategy is optimal in the worst-case up to logarithmic factors. A Monte Carlo strategy can use randomization and is allowed to err with bounded probability. In particular, we prove that randomized strategies outperform deterministic strategies. In theory, the Monte Carlo backtracking strategy outperforms all strategies currently used in practice in the worst-case. Further theoretical results include a characterization of the structure of the aforementioned backtracking trees and an analysis of design choices in the so-called color refinement algorithm, a crucial subroutine used by the solvers.

In turn, the design of DEJAVU is based on these theoretical results. In particular, the backtracking strategy of DEJAVU follows the near-optimal Monte Carlo strategy. The solver further contains various novel practical components. These components are carefully designed to complement the Monte Carlo strategy. Notably, one of the components is a preprocessor designed to shrink large and sparse inputs, which can also be used with all the other state-of-the-art solvers. Benchmarks on a vast library of graphs reveal that DEJAVU is faster than any other state-of-the-art solver on most tested graph classes.

Zusammenfassung

Die Ausnutzung von Symmetrien hat einen großen Einfluss auf die Effizienz praktischer Algorithmen in vielen verschiedenen Bereichen. Egal wo und wie Symmetrien verwendet werden, wird eine effiziente Möglichkeit benötigt, um Symmetrien zunächst zu berechnen. Dafür werden kombinatorische Strukturen in der Praxis üblicherweise als Graphen modelliert. Die Automorphismengruppe dieses Modellgraphen entspricht dann den Symmetrien der kombinatorischen Struktur. Ein Graphisomorphie-Algorithmus wird anschließend verwendet, um die Automorphismengruppe des Modellgraphen zu berechnen. Praktische Graphisomorphie-Algorithmen werden seit über 50 Jahren entwickelt. Obwohl alle moderne praktische Algorithmen auf demselben Prinzip basieren, nämlich dem Individualization-Refinement (IR) Paradigma, gibt es dennoch große Unterschiede zwischen den Implementierungen. Dies wiederum führt zu großen Leistungsunterschieden zwischen den Algorithmen. Obwohl die praktische Graphisomorphie schon oft als "gelöst" bezeichnet wurde, ist in der Tat kein einziger praktischer Algorithmus in der Lage, alle Graphen aus den verschiedenen Anwendungsbereichen effizient zu lösen. Es gibt sogar einzelne Anwendungen, in denen es keiner der Algorithmen schafft, alle Graphen effizient zu lösen. Idealerweise gäbe es einen einzigen Algorithmus, der auf allen Graphen schnell ist.

Die vorliegende Arbeit beschäftigt sich mit dem Entwurf eines neuen Algorithmus, DEJAVU, für die Erkennung von Symmetrien auf Graphen. Anstatt ad-hoc nach einer besseren Implementierung zu suchen, führen wir eine theoretische Analyse der wesentlichen algorithmischen Bestandteile des IR Paradigmas durch. Ein zentrales Ergebnis ist ein theoretisches Modell für die Backtracking-Strategien moderner praktischer Algorithmen. Wir beweisen, dass ein bestimmter Monte-Carlo Algorithmus innerhalb des Modells optimal bis auf logarithmische Faktoren ist. Ein Monte-Carlo Algorithmus kann Zufall in Berechnungen miteinbeziehen, und darf mit einer begrenzten Wahrscheinlichkeit Fehler machen. Insbesondere beweisen wir, dass in unserem Modell probabilistische Strategien beweisbar besser sind als deterministische. Die Monte-Carlo Strategie ist asymptotisch schneller als alle anderen Strategien, die in der Praxis verwendet werden. Weitere theoretische Resultate beinhalten eine konstruktive Charakterisierung der Backtracking-Bäume, sowie eine Analyse von Design-Entscheidungen im sogenannten Color Refinement Algorithmus.

Das Design von DEJAVU basiert auf unseren neuen theoretischen Erkenntnissen. Insbesondere verwendet DEJAVU eine Monte-Carlo Backtracking-Strategie. Darüber hinaus besteht der Solver aus vielen weiteren praktischen Komponenten, die dafür entwickelt wurden, die Monte-Carlo Strategie zu unterstützen. Eine dieser Komponenten ist ein Preprocessor, der auch mit allen anderen modernen Implementierungen verwendet werden kann. Eine experimentelle Analyse auf einer weitreichenden Kollektion von Graphen demonstriert anschließend, dass DEJAVU auf der großen Mehrheit der Graphklassen signifikant schneller ist als alle anderen Implementierungen.

Acknowledgements

I thank my colleagues Jendrik Brachter, Sofia Brenner, Billy Joe Franks, Gaurav Rattan, and Florian Wetzels. Our collaborations were truly the most fun part of the journey. I also want to thank Moritz Lichter, Georg Schindling, Thomas Schneider, and Lena Volk for proofreading parts of this thesis and for being amazing colleagues. I thank Marc Pfetsch and Christopher Hojny for sharing their insights on symmetry breaking with me. I thank Brendan McKay and Adolfo Piperno for the inspiring discussions on practical graph isomorphism, which always resulted in additional tinkering with the implementation. I am grateful to Pascal Schweitzer for teaching me an endless number of things.

Last but certainly not least, I thank Luisa for always having my back and supporting me in all of my endeavors.

Contents

Chapter 1

Introduction

1.1 Motivation

Symmetry is an intuitive concept that appears in various contexts, both in nature and mathematics. Indeed, skilled mathematicians often argue "by symmetry". This can often severely shorten a proof or argument. A particular kind of symmetry is one exhibited by combinatorial structures. For example, parts of a computational problem or variables in an equation may be interchangeable on a syntactic level: exchanging x and y in $x^2+2xy+y^2$ leaves us with the same formula. We intuitively know that when arguing over the variables x and y, we only need to worry about one of the symmetrical choices. It has often been observed that such symmetries are commonplace and naturally occur through mathematical modeling [38]. Indeed, instances of computational problems can exhibit a remarkable amount of symmetry [92, 101]. Naturally, we therefore want algorithms also to be able to argue "by symmetry".

In an algorithmic context, systematically reducing symmetrical computations can lead to a tremendous gain in efficiency [51, 26]. Indeed, the use of symmetry has a dramatic impact on the efficiency of practical algorithms in various fields. This includes computer graphics [68], automated reasoning [53], machine learning [105], chemistry [95, 99, 77], constraint programming in general [42], SAT solvers [101], software verification [22, 67], model checking [44, 79], SMT solvers [30], mixed integer linear programming [72, 92, 54], answer set programming [32] and many more. Precisely how symmetries are used best in an algorithmic context remains a matter of active research. Even for a particular application, such as SAT, there may be numerous approaches which try to balance computational overhead with the strength of symmetry reduction [26, 2, 34, 100, 35, 24, 33, 78, 52, 57, 64].

However, no matter how symmetries are used precisely, there needs to be a way to obtain them. If the goal is to speed up other computations, being able to *efficiently* obtain symmetries is essential. Fortunately, computing the symmetries of a combinatorial object, such as an instance of a computational problem, can very often be efficiently reduced to computing the symmetries of a graph [80]: research can be focused on efficiently computing the symmetries of graphs.

The efficient detection of symmetries of graphs is the topic of this thesis. Computing symmetry is intimately related to the so-called graph isomorphism problem. Hence, practical algorithms for symmetry detection are often referred to as *practical graph isomorphism algorithms*. In the early days, the development of these algorithms attracted so much research that it was dubbed a disease [97]. Half a century of further research has led to the current suite of state-of-the-art algorithms. The thesis consists of an algorith-

mic analysis of ingredients used in practical graph isomorphism algorithms, which then inspires the design of a new, significantly more powerful implementation called DEJAVU.

Before going into more detail, let us begin our exposition with more background on using symmetries in practice. This will then lead us to the current state-of-the-art in symmetry detection.

Symmetry in Practice. How symmetries are used best in an algorithmic context remains a matter of active research. While there are many different application areas, quite commonly, symmetries are used to speed up an exponential-time backtracking search. Here, the difficulty lies in the fact that methods need to balance computational overhead with the strength of symmetry reduction. Indeed, it should be mentioned that there can be several sources for overhead: it can stem from the symmetry detection, the computational overhead of the symmetry reduction method, as well as interfering with other strategies used by the original backtracking algorithm. It is possible to remove *all* symmetries from a backtracking search using so-called isomorph-free exhaustive generation (see e.g., [75, 57]). However, these techniques incur a substantial computational overhead. Another natural strategy is to include a symmetry-aware backtracking rule, which essentially skips symmetric choices (see e.g., [89, 100, 33]). A downside of such techniques is that they usually require a substantial modification of the underlying algorithm, which may interfere with other strategies and may therefore be undesirable. In constraint programming, another very common technique is to add *static symmetry breaking constraints* to a given instance. This adds additional variables and constraints which ought to restrict the search space to only asymmetric solutions (see e.g., [26, 37, 2, 34, 92]). While symmetry reduction is usually limited, a distinct advantage of static symmetry breaking constraints is that the underlying algorithms do not have to be modified.

Computing Symmetry. All these numerous applications and approaches to exploiting symmetry do however have one issue in common: before symmetries of a structure can be exploited, one first has to have algorithmic means to find them. Towards this goal, a typical first step is that the given combinatorial structure is transformed into an annotated graph whose symmetries correspond to the symmetries of the original structure. Indeed, this is efficiently possible for all finite relational structures [80].

The approach is also used in practice: the instances of a computational problem are transformed into a so-called *model graph*, which is then given to a tool solving symmetry detection for graphs. Figure 1.1 illustrates such a model graph for a Boolean formula, as it might occur in SAT. First, every variable and its negation are a vertex of the graph. Second, every disjunction of the formula is a vertex connected to the contained variables or negated variables. The symmetries of this graph correspond precisely to the syntactic symmetries of the formula [101].

It thus suffices to be able to compute the symmetries of a given graph. In fact, even simple, undirected graphs suffice [80]. One may wonder: *how difficult is this problem?* It turns out that computing the symmetries, or the *automorphism group*, of a given graph is polynomial time equivalent to the graph isomorphism problem.

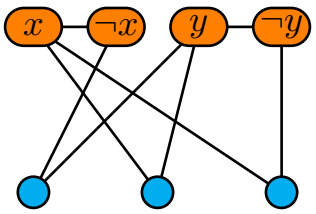


Figure 1.1: A model graph for the Boolean formula $(x \vee y) \wedge (\neg x \vee y) \wedge (\neg y \vee x)$.

Whether the graph isomorphism problem can be solved in polynomial time is a long-standing open problem [12]. Resolving this question even seems difficult for the restricted subcase of the group isomorphism problem [12], for which it seems particularly challenging to apply the combinatorial machinery used elsewhere [19]. Quite trivially, we know that the graph isomorphism problem can be solved in nondeterministic polynomial time (i.e., NP). Moreover, we know that graph isomorphism is in coAM [15], which means there is a randomized protocol for verifying graph non-isomorphism. NP-completeness of graph isomorphism is arguably implausible: firstly, since this would lead to a collapse of the polynomial hierarchy [46, 15]. Secondly, the best-known theoretical algorithm, due to Babai [12], runs in quasi-polynomial time. Hence, NP-completeness would immediately contradict the strong exponential time hypothesis [21].

On the other hand, graph isomorphism is easy on average: asymptotically almost all graphs can be solved using a naive linear time algorithm [14]. Furthermore, numerous restricted graph classes are known for which graph isomorphism can be solved in polynomial time. This includes trees [1], planar graphs [56], graphs with bounded degree [71], bounded tree width [18], bounded rank width [50], graphs with a forbidden minor [48], tournaments with bounded twin width [49], and many more. Many of these results use a group-theoretic framework due to Luks [71]. As already pointed out by McKay and Piperno [76], while some of these algorithms have practical merit [1, 56], most of them use subroutines which do not seem to be viable. Indeed, in practice, a different class of algorithms is used.

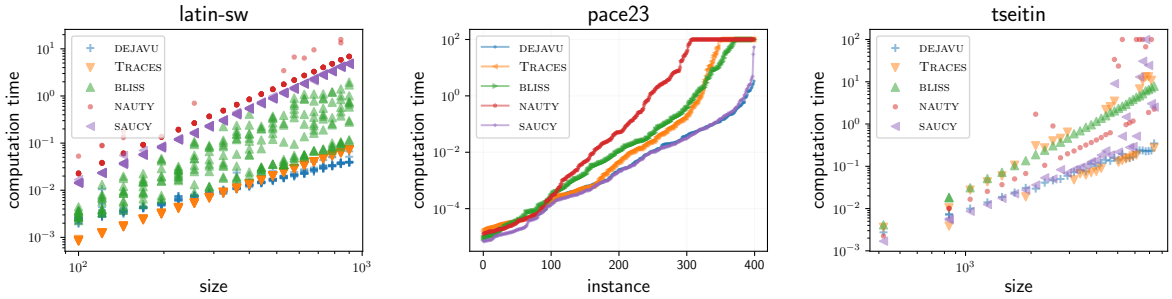
Practical Graph Isomorphism. The current state-of-the-art implementations for computing the symmetries of graphs are BLISS [58, 59], NAUTY [74, 76], TRACES [93, 76, 94], as well as SAUCY [27, 28, 60, 23]. It should be mentioned that NAUTY, TRACES, and BLISS can also solve a more general task: they can compute a canonical labeling, which provides a canonical representative within an isomorphism class of graphs.

All these tools are based on the so-called *individualization-refinement* (IR) paradigm. Parris and Read introduced the IR paradigm in a series of papers [91]. The paradigm was then further developed in publications which predate NAUTY [25, 9]. However, NAUTY was the first implementation to truly realize the potential of IR, with the introduction of several essential ingredients [74], further explained below. The paradigm itself essentially describes how the backtracking trees of the tools are constructed, as is explained in the following. IR consists of two main procedures: a *refinement* routine and the *individualization* technique. A refinement is a heuristic to distinguish vertices that can not be

mapped by symmetry: it partitions the vertices so that for two vertices of two different parts, no symmetry maps one to the other. All solvers implement *color refinement*, also known as the 1-dimensional Weisfeiler-Leman algorithm. The routine first distinguishes vertices according to their degree, then their neighbors' degrees, and so forth. In IR, vertices not yet distinguished into separate partitions by the refinement are artificially separated using individualization. The algorithm chooses a partition, or "selects a cell", and separates one of the vertices into its own partition. More precisely, a branch in the backtracking tree is created for every vertex of the selected cell. Then, the refinement is applied to each branch, and the process repeats. We should note that color refinement and the more general k-dimensional Weisfeiler-Leman algorithm are also studied in various other domains. In the context of machine learning, graph neural networks are strongly related to Weisfeiler-Leman, and various results try to exploit this connection [83, 39]. Moreover, aspects of the Weisfeiler-Leman algorithm are strongly related to finite model theory [20].

Algorithms in the IR paradigm have also been the subject of theoretical study. Goldberg showed that with certain modifications, IR algorithms can achieve single exponential runtime [45]. On the other hand, graphs guaranteed to cause exponential runtime for IR algorithms are rare and difficult to obtain. Indeed, it is known that color refinement asymptotically distinguishes almost all graphs [14]. Based on the construction of [20], Miyazaki was the first to provide graphs on which NAUTY provably exhibited exponential runtime [82]. The graph construction makes use of a particular "cell selector" used by NAUTY, and could therefore be easily circumvented [108]. This was later rectified by Neuen and Schweitzer, proving an exponential lower bound regardless of the cell selector used [87].

While all state-of-the-art tools for symmetry detection are based on the IR paradigm, there are still vast differences. In particular, there are differences in the pruning techniques and how tools traverse the IR backtracking trees. Let us now briefly introduce the tools and some of their differences. Designed by McKay in 1977, NAUTY had been the indisputable fastest solver for decades. A key feature introduced by NAUTY was *automorphism pruning*, meaning the tool itself made use of symmetry during search [74]. Furthermore, NAUTY introduced a particularly efficient implementation of color refinement [74], which is still used as the blueprint for all modern implementations. Continuously updated over several decades, NAUTY now features a version for dense graphs, one for sparse graphs, a vast array of invariants, and the Schreier-Sims algorithm for more rigorous automorphism pruning [76]. The tool SAUCY, published in 2004, was initially designed to exploit symmetry in SAT formulas [27] to be used in SAT solvers. SAUCY can exploit sparsity both in the input and output: a crucial feature is that SAUCY attempts to detect and output symmetries with small support efficiently. Indeed, graphs stemming from SAT formulas often exhibit symmetries with small support. The tool BLISS, published in 2007, constitutes a reimplementation of the algorithm underlying NAUTY with a particular focus on efficient low-level data structures. Besides efficient low-level data structures, BLISS introduced a way of dealing with homogeneously connected components of graphs and a form of conflict propagation [59]. All the tools mentioned above follow a natural *depth-first* approach to traversing the IR backtracking tree. The tool TRACES



(a) Graphs stemming from Latin squares.

(b) Graphs from the PACE 2023 competition.

(c) Graphs from "Tseitin" SAT instances.

Figure 1.2: Performance of state-of-the-art practical graph isomorphism solvers on select benchmark families. The timeout is 100 seconds.

by Piperno, released in 2008, broke away from this principle and pioneered a completely different search strategy for IR trees. TRACES mainly uses *breadth-first* search in conjunction with *random root-to-leaf walks* of the IR backtracking tree. This strategy excels at effectively pruning the search space of difficult combinatorial graphs. Moreover, the tool contains many new features, including novel pruning techniques, a highly engineered color refinement implementation, and a limited form of preprocessing.

Overall, the reader might now wonder: *which tool is the fastest one?* Being particularly efficient on difficult combinatorial graphs, TRACES has a strong case for being the fastest overall solver. This is exemplified by the performance of TRACES on graphs obtained from Latin squares, as illustrated in Figure 1.2a. However, this is not universally true for all graph classes. The diverging techniques used by the solvers lead to significant differences in their performance across different graph classes. In particular, there are essential practical graph classes where SAUCY runs faster than TRACES. For example, on graphs stemming from the PACE challenge 2023 [90] SAUCY can utilize its strongly tailored detection of sparse symmetries, as can be seen in Figure 1.2b. There are also cases in which NAUTY or BLISS are faster. Lastly, there are classes where all solvers are intractable, and TRACES seems particularly slow: the so-called shrunken multipedes provably incur exponential runtime for any IR algorithm [86, 87].

Only having solvers available that are geared toward specific types of graphs is an undesirable situation. A practitioner looking for a symmetry detection tool needs to check which tool suits the application best. Even worse, there might not even be one single tool that suits all the encountered graphs in an application. This is true in SAT solving: designed with SAT in mind, SAUCY is the fastest of these tools on most SAT competition instances. However, SAUCY struggles to solve, for example, model graphs from the well-established tseitin benchmark family [109] efficiently, as illustrated in Figure 1.2c. We should also note that the state-of-the-art symmetry breaking tool in SAT describes symmetry detection as the bottleneck [34] and artificially limits the time spent on symmetry detection. Regarding our overall discussion, solvers will also naturally struggle with input graphs that are combinations of the different types of graphs.

Ideally, one would like to have a single implementation that is fast on all graphs.

However, it seems that to create an algorithm that combines the benefits of the different solvers, a deeper understanding of the various strategies and techniques is necessary.

1.2 Contributions

In this thesis, we describe the design of a new symmetry detection tool called DEJAVU. Instead of trying to develop a better implementation ad hoc, we first perform a thorough theoretical analysis of essential algorithmic ingredients used in IR solvers. In particular, we give a theoretical analysis of the *IR backtracking behavior*, as well as the *color refinement* algorithm. The most crucial result concerns a theoretical model for IR traversal strategies, which essentially captures how algorithms traverse the IR backtracking trees. In this model, we prove that randomized strategies outperform deterministic ones. We show that a specific Monte Carlo strategy is optimal in the worst-case up to logarithmic factors. In turn, we base the design of DEJAVU precisely on this near-optimal Monte Carlo strategy. This is augmented with an extensive array of novel practical techniques and novel applications of successful methods used in state-of-the-art solvers. Benchmarks demonstrate that this yields a significantly more powerful implementation compared to the state-of-the-art. A detailed summary of the individual contributions follows below.

Traversal Strategy. Looking at the history of state-of-the-art IR tools, algorithms traditionally followed a depth-first search approach to traverse the search tree. However, TRACES broke away from this principle, performing a form of breadth-first search that is combined with a random traversal of the tree. Practical observations demonstrate that the approach is particularly effective on difficult, combinatorial graphs. But this immediately raises the question: *are there theoretical, structural reasons why this traversal strategy is favorable?*

We design a theoretical model for the traversal strategy of IR algorithms. The model is based directly on trees, which ought to model the IR backtracking trees of the solvers. In this model, we investigate lower and upper bounds within three settings: deterministic, Monte Carlo, and Las Vegas. In the Monte Carlo setting, algorithms may incorporate randomization into their computations and may err with bounded probability. In the Las Vegas setting, algorithms may use randomization but must not err.

Regarding the bounds, for deterministic algorithms, we prove matching linear lower and upper bounds. The matching bounds even hold in the case of binary trees. In the worst-case, deterministic algorithms need to essentially investigate the entire IR backtracking tree. For Las Vegas algorithms, we prove a linear lower and upper bound for unrestricted IR trees. In the case where IR trees have a bounded degree d, we prove a $\mathcal{O}(d \log(N)\sqrt{n})$ upper bound for expected worst-case runtime. (Here, n and N refer to the size of the involved trees; see Chapter 3 for more details.) For Monte Carlo algorithms, we show a lower bound of $\Omega(\sqrt{n})$ and an almost matching upper bound of $\mathcal{O}(\log(n)\sqrt{n})$. This means that even in the worst-case, there is a Monte Carlo traversal strategy that only ever investigates a $\mathcal{O}(\log(n)\sqrt{n})$ portion of the IR backtracking tree.

Furthermore, we provide a constructive characterization of IR trees. We provide necessary conditions for a tree to be an IR tree using color refinement. Then, for each tree that satisfies the necessary conditions, we construct a graph and a so-called cell selector, which lead to an IR tree that structurally matches the input tree. Specifically, this proves that the IR trees used in our lower bound constructions can stem from actual inputs to the algorithm.

Overall, the results explain why the randomized breadth-first with experimental path search strategy of Traces is often superior to the depth-first search employed by other tools. The strategy employed by Traces incorporates aspects of both the Monte Carlo and Las Vegas strategies. Yet, the worst-case performance of Traces is still linear. Our Monte Carlo strategy and Las Vegas strategy asymptotically outperform all traversal strategies that are currently used in practice in the worst-case. In our model, the Monte Carlo strategy is near-optimal in the worst-case. It is provably more efficient than any Las Vegas or deterministic strategy.

Color Refinement. Color refinement is continuously applied during the execution of an IR algorithm and is thus the most important subroutine. The efficiency of the color refinement implementation is a crucial factor in the overall runtime of a solver. Furthermore, advanced strategies used by the solvers are sometimes entangled with the color refinement algorithm itself. The color refinement algorithm is a crucial cornerstone in the design of an efficient IR algorithm. Color refinement can be implemented such that it runs in worst-case time $\mathcal{O}((n+m)\log n)$ [16]. Indeed, within a model of modest assumptions, there is a matching $\Omega((n+m)\log n)$ lower bound [16]. It should be noted that the theoretical algorithm used for the upper bound in [16] significantly differs from the implementations used by modern solvers.

We describe a color refinement algorithm that closely resembles the implementation used in dejavu and Traces. Then, we argue its correctness and worst-case runtime of $\mathcal{O}((n+m)\log n)$. This means the practical implementation also matches the theoretical upper bound [16]. Based on this exposition, we describe crucial engineering tricks and other strategies based on color refinement, as used by contemporary solvers.

The efficient implementation of color refinement *splits* partitions of vertices, called color classes, according to their neighbor counts in other color classes. A central design choice of the algorithm is the *order* in which these splits are performed. The splits are usually kept in a *worklist*, such as a stack or queue, the choice of which determines the order of the splits. Indeed, the order implied by the worklist is of no importance for the worst-case runtime, or the lower bound construction. Still, it could be that one worklist is always as good or superior to another. There might even be a worklist that is always competitive with any other worklist on all graphs. We provide a rigorous theoretical analysis of the worklist. We define an online model and an offline model, enabling us to formally compare and analyze different worklist choices in color refinement. The results within these two models concur: in the online model, no worklist that is competitive beyond a logarithmic factor to all other worklists can exist. In the offline model, unless $\mathsf{P} = \mathsf{NP}$, the optimal worklist can not be approximated in polynomial time up to a logarithmic factor.

Solver Design. We describe the design of the DEJAVU algorithm for symmetry detection. As informed by our theoretical analysis, the foundation of the solver is a Monte Carlo algorithm. The algorithm has a *one-sided bounded error*: the user provides a probability ϵ, and in each run, the solver may *miss a symmetry* with a probability of at most ϵ. In particular, the solver never outputs a permutation that is *not* a symmetry of the input graph. In practice, it turns out that applying *only* the Monte Carlo algorithm does not make for a good solver. The algorithm cannot efficiently solve numerous graph classes: recall that the Monte Carlo strategy is only proven superior *in the worst case*. In many cases, IR trees are either very easy or can be readily pruned. In order to deal with these cases more efficiently, the solver incorporates a multitude of further components. The aim is to efficiently reduce graphs and IR trees to a difficult core, which is then passed to the Monte Carlo strategy. This includes many novel techniques and novel applications of existing methods. We highlight the following key features of the solver:

- A *Monte Carlo* search, which can be mixed with both *depth-first* and *breadth-first* search.
- A *universal preprocessor*, which is designed to shrink and simplify the input graphs. It reduces vertices with degrees at most 2, twins, and features further reductions based on the so-called quotient graph. The preprocessor can be used in conjunction with all state-of-the-art tools.
- A *restart-and-inprocessing scheme*, which draws inspiration from the design of contemporary combinatorial solvers. This means whenever a graph is difficult, the solver occasionally restarts its search and varies the strategies used. On each restart, partial results are used to simplify the graph.
- A *compressed Schreier structure*. The Schreier structure is used to keep track of the collected symmetries. The structure is compressed using information gathered from the IR tree.
- The so-called *trace deviation set* pruning technique, which is designed to speed up breadth-first search in IR trees.

Juggling these different components and aspects from different areas poses a severe challenge. Components are only valuable as part of a coherent strategy, which must also be implemented in a highly efficient manner.

We describe how the DEJAVU algorithm combines these components. We provide correctness arguments for each component individually and essential interactions between the components. Furthermore, we prove that the Monte Carlo strategy implicitly performs *complete automorphism pruning*, meaning it takes full advantage of the presence of symmetry.

Benchmarks. We compare DEJAVU to all state-of-the-art symmetry detection tools across a wide range of graph classes. These graph classes include almost all graphs

listed in the de-facto default benchmark library [85] as well as further graph classes encountered in practice. The benchmarks demonstrate that DEJAVU is the fastest solver on the majority of graph classes (33 out of 44). A solver is *competitive* on a graph class whenever it is the fastest solver or if it does not take more than twice the time of the fastest solver. DEJAVU is competitive on almost all graph classes (39 out of 44). The second-best solver in our benchmarks is TRACES, which is competitive on half of all classes (22 out of 44). Whenever the results of DEJAVU could be compared against the results of a deterministic solver, we checked whether DEJAVU missed any symmetries due to the Monte Carlo algorithm. No probabilistic error was observed in our benchmarks.

Moreover, the preprocessor of DEJAVU can be used in conjunction with any other state-of-the-art solver. We demonstrate that the preprocessor significantly speeds up NAUTY, SAUCY, BLISS, and TRACES on the benchmark library.

Barring the potential one-sided error, DEJAVU is probably the most powerful symmetry detection algorithm currently in use.

1.3 Structure of this Thesis

We give an overview of the contents of each chapter.

Chapter 2. We begin by introducing important preliminaries. This includes basic notation for graphs, data structures, and algorithms. The main part of the chapter concerns individualization-refinement, the paradigm that all state-of-the-art symmetry detection tools follow. The chapter concludes with a description of all state-of-art symmetry detection tools.

Chapter 3. Next, we discuss how tools traverse their IR search tree. In this chapter, we describe a model for search tree traversal. We provide upper and lower bounds within the model across different settings. Furthermore, a characterization of IR trees is given. The analysis of traversal strategies is joint work with Pascal Schweitzer, and has been previously published in [6]. The characterization of IR trees is joint work with Jendrik Brachter and Pascal Schweitzer, and has been previously published in [3].

Chapter 4. Color refinement is the most crucial subroutine of symmetry detection tools. In this chapter, a detailed description of a modern implementation is provided. We discuss several optimization strategies employed by the different tools. The chapter also includes an analysis of the competitiveness of different "worklists" in the algorithm. The analysis of worklists has been previously published in [8], and is joint work with Pascal Schweitzer and Florian Wetzels.

Chapter 5. Next, we describe a preprocessor for symmetry detection. The preprocessor focuses on strategies to reduce low-degree vertices, twins, and related strategies based on the color refinement algorithm. The preprocessor was previously published in [7], and is joint work with Pascal Schweitzer and Julian Stieß.

Chapter 6. All these previous results culminate in our description of the DEJAVU algorithm. The chapter describes all the components of the algorithm as well as the high-level procedure. Furthermore, we argue the correctness of the components as well as important interactions between the components. Previous iterations of the solver have been published in [5, 4], which is joint work with Pascal Schweitzer. We note that the implementation of the solver described in this thesis differs significantly from previous publications.

Chapter 7. We benchmark the resulting implementation against all other state-of-the-art solvers. Furthermore, we also test the effectiveness of the preprocessor in conjunction with all state-of-the-art solvers.

Chapter 8. Lastly, we discuss interesting theoretical open questions, as well as how symmetry detection may be further improved in future work.

It should be noted that the introduction and preliminaries also contain parts of the previous publications [6, 3, 8, 7, 5, 4].

This exposition aims to serve as a reference for how to construct an efficient IR algorithm. Of course, a major part of this concerns our theoretical results. However, there are also many crucial practical details. In particular, in Chapter 2 and Chapter 4, we provide low-level descriptions for crucial procedures and data structures used in IR algorithms. In order to keep things manageable, we skip these low-level descriptions for high-level algorithms and ideas, such as in Chapter 5 and Chapter 6. If the reader desires more algorithmic detail in certain places, in light of our descriptions here, the remaining information can hopefully be easily gathered from the implementation itself. The implementation is open source and freely available [31].

Chapter 2

Preliminaries and Related Work

We begin by introducing basic notation for graphs and symmetries. This includes a definition of the graph isomorphism problem, graph automorphism problem, and canonical labeling problem. We reason why these problems are of interest to other areas of mathematics and computer science. Considering the graph automorphism problem, a basic fact is that the automorphisms of a graph form a permutation group. Therefore, we introduce a few rudimentary results of computational group theory. We then describe several basic algorithms and data structures needed for symmetry detection.

This culminates in the description of the *individualization-refinement* (IR) framework, which is the foundation of all state-of-the-art practical graph isomorphism algorithms. We describe the various ingredients of the framework. Then, we explain how IR is used to compute graph isomorphisms, graph automorphisms, and canonical labelings. Lastly, we give a brief description and history of all the state-of-the-art practical graph isomorphism solvers, with a particular focus on techniques for symmetry detection.

2.1 Graphs and Groups

2.1.1 Graphs

A finite graph $G = (V, E)$ consists of a finite set of vertices $V \subseteq \mathbb{N}$ and an edge relation $E \subseteq V \times V$. Unless stated otherwise, we assert the following three properties about a graph:

1. The set of vertices V is $\{0, \dots, n-1\}$ and $m \coloneqq |E|$ denotes the number of edges.
2. The edge relation contains no *self-loops*, which means there exists no $v \in V$ for which $(v, v) \in E$.
3. Graphs are *undirected*, which means E is symmetric.

If E is not required to be symmetric, the graph is called *directed*. Furthermore, we say that $n + m$ denotes the *size* of a graph. For the sake of convenience, we may refer to the set of vertices of G with $V(G)$, and to the set of edges with $E(G)$.

We call two vertices $v_1 \in V, v_2 \in V$ *adjacent*, if $(v_1, v_2) \in E$ holds. Another way to phrase this is to say that v_1 and v_2 are *neighbors* in G. For a vertex $v \in V$, the *open neighborhood*

$$N(v) \coloneqq \{v' \in V \mid v' \text{ is adjacent to } v\}$$

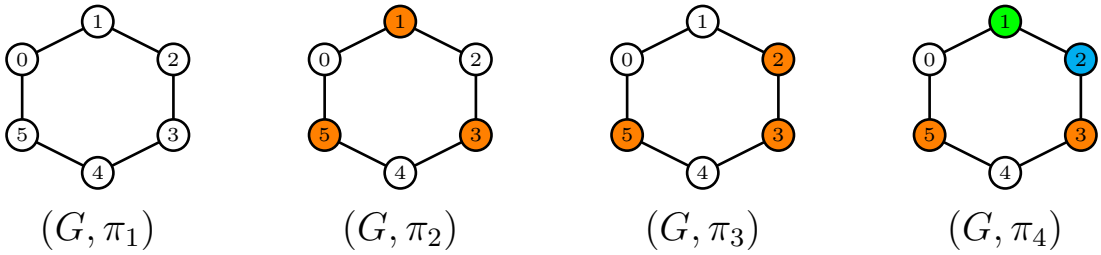


Figure 2.1: A graph on 6 vertices with three different vertex colorings. The vertex coloring π_2 is finer than π_1, and π_3 is finer than π_1. The vertex colorings π_2 and π_3 are incomparable using the finer relation. The coloring π_4 is finer than all the other colorings.

is the set of all its adjacent vertices. The *closed neighborhood* additionally includes a vertex in its own neighborhood, denoted by

$$N[v] \coloneqq N(v) \cup \{v\}.$$

The *degree* $\deg(v) \coloneqq |N(v)|$ of a vertex is the size of its open neighborhood. For a set of vertices $V' \subseteq V(G)$ the *neighborhood* is the set $N[V'] \coloneqq (\cup_{v \in V'} N(v)) \setminus V'$. An edge $(v_1, v_2) \in E$ is called *incident* to its *endpoints* v_1 and v_2. A vertex $v \in V$ is *incident* to all edges where v appears as an endpoint.

Let $V' \subseteq V(G)$ denote a subset of the vertices of a graph G. We define the *induced subgraph* $G[V']$. The vertex set is $V(G[V']) \coloneqq V'$ (observe that this may in particular not be $\{0, \dots, n-1\}$). The edge relation consists of all the edges where both endpoints are in V'. Formally, let

$$E(G[V']) \coloneqq E(G) \cap (V' \times V').$$

An important notion is to be able to *color* a graph. More specifically, we are coloring the vertices of a graph. Formally, a *vertex coloring* $\pi \colon V \to \{0, \dots, n-1\}$ maps vertices of a graph to *colors* $0, \dots, n-1$. We call $\pi^{-1}(i) \subseteq V$ with $i \in \{0, \dots, n-1\}$ and $\pi^{-1}(i) \neq \emptyset$ a *cell*, or *color class*, of π. If $\pi^{-1}(i) = C = \{v\}$, we call v a *singleton*, and C a *singleton color class*. With

$$|\pi| \coloneqq |\{i \mid i \in \{0, \dots, n-1\} \wedge |\pi^{-1}(i)| \geq 1\}|$$

we denote the *number of cells* (or color classes). If $|\pi| = n$ holds, we call π *discrete*. Note that a discrete coloring is a permutation of $\{0, \dots, n-1\}$. In other words, a discrete coloring characterizes a permutation on the set of vertices V. A *vertex-colored* graph (G, π) simply consists of a graph and a vertex coloring.

In the literature, vertex colorings are also often called *ordered partitions*. Observe that a vertex coloring indeed partitions the set of vertices V according to their color. We refer to the partitioning induced by a vertex coloring π as the *color partition* of π. Each color class of a coloring π is a part of the color partition. Since the colors themselves are ordered, we can order the parts according to the order of the colors.

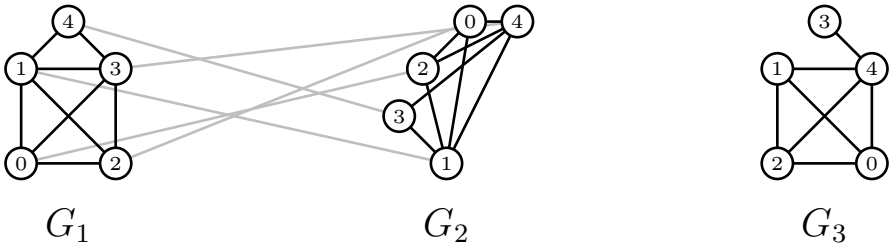


Figure 2.2: Two isomorphic graphs $G_1 \cong G_2$, and another graph G_3 that is not isomorphic to G_1 and G_2. Note that G_3 has a vertex of degree 1, whereas G_1 and G_2 have no vertex of degree 1.

We want to be able to compare two given vertex colorings. A vertex coloring π is *finer* than another coloring π', if

$$\pi(v) = \pi(v') \implies \pi'(v) = \pi'(v')$$

holds for all $v \in V, v' \in V$. This means that any two vertices which have the same color in π', must also have the same color in π. We denote this with $\pi \preceq \pi'$. Going into the other direction, we call π' *coarser* than π.

Figure 2.1 illustrates three different vertex colorings of a graph. We note how $\pi_2 \preceq \pi_1$ and $\pi_3 \preceq \pi_1$ hold, whereas π_2 and π_3 are incomparable using $\preceq$. Furthermore, $\pi_4 \preceq \pi_i$ for all $i \in \{1, 2, 3\}$.

2.1.2 Isomorphisms and Automorphisms

We now introduce the concepts of isomorphisms and automorphisms. Fundamentally, an isomorphism describes whether two graphs are structurally equivalent when ignoring the names of the vertices. As such, it defines a notion of structural equivalence.

We define the shorthand notation $(x, y)^\varphi := (x^\varphi, y^\varphi)$ and $S^\varphi := \{s^\varphi \mid s \in S\}$. Formally, two graphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ are said to be *isomorphic*, whenever there exists a bijection $\varphi : V_1 \to V_2$ such that

$$\varphi(G_1) = (V_1^\varphi, E_1^\varphi) = (V_2^\varphi, E_2^\varphi) = G_2.$$

We call φ an *isomorphism* between G_1 and G_2. We may also write $G_1 \cong G_2$ to denote that G_1 and G_2 are isomorphic.

Figure 2.2 shows two isomorphic graphs G_1 and G_2, and a corresponding isomorphism (gray lines). Note that every isomorphism must preserve the neighborhood of each vertex. Hence, quite naturally, vertices of differing degrees can not be mapped onto each other. Since G_3 has a vertex of degree 1, whereas G_1 and G_2 contain no vertex of degree 1, there can not be an isomorphism between G_3 and the other two graphs.

We call two vertex-colored graphs (G_1, π_1) and (G_2, π_2) isomorphic whenever there exists an isomorphism φ between G_1 and G_2, which additionally only maps vertices of

the same color onto each other. Formally, φ must additionally satisfy the property

$$\forall v \in V_1 : \pi_1(v) = \pi_2(v^\varphi).$$

We define a corresponding computational problem:

Problem 1 (Graph Isomorphism)**.** *Given two graphs G_1 and G_2, does $G_1 \cong G_2$ hold?*

Based on this, we define the automorphisms or symmetries of a graph. The set $\mathrm{Aut}(G)$ denotes the *automorphism group* of a graph G. The automorphism group $\mathrm{Aut}(G)$ contains all permutations φ of the vertices $V(G)$ that are isomorphisms, i.e., a bijective map $\varphi\colon V \to V$ where

$$G^\varphi \coloneqq (V^\varphi, E^\varphi) = (V, E) = G.$$

In other words, automorphisms are isomorphisms mapping the graph to itself. Again, for vertex-colored graphs, we additionally require that the vertex coloring is preserved. We thus define the colored automorphism group $\mathrm{Aut}(G, \pi)$ as those permutations φ which satisfy $(G, \pi)^\varphi = (G^\varphi, \pi^\varphi) = (G, \pi)$. (Let $\pi^\varphi \coloneqq \pi \circ \varphi$.)

Again, we define a corresponding computational problem. It uses the notion of generating sets, which is formally defined further below (in Section 2.1.3):

Problem 2 (Graph Automorphisms)**.** *Given a graph G, determine a generating set S such that $\langle S \rangle = \mathrm{Aut}(G)$.*

Practical graph isomorphism algorithms often solve another task, namely computing *canonical labelings* of graphs. While not the main topic of this thesis, the problem is strongly related to computing graph isomorphisms and automorphisms. The idea is to compute a canonical form of a graph. Our definition follows the one given in [76]. Let $\mathcal{G}$ denote the set of all graphs. We call $C : \mathcal{G} \to \mathcal{G}$ a canonical form, whenever for all $G \in \mathcal{G}$ and all $\varphi \in \mathrm{Sym}(V(G))$ the following hold:

1. All graphs isomorphic to G result in the same canonical form, which means $C(G^\varphi) = C(G)$.
2. The canonical form is isomorphic to the input graph, which means $C(G) \cong G$.

(The definition assumes that all graphs on n vertices are defined using the same vertex set.) Again, we can define a similar vertex-colored version, where colors need to be preserved. A *canonical labeling* is a canonical form that is achieved by permuting the vertices of the graph. A corresponding, natural computational problem is the following:

Problem 3 (Canonical Labeling)**.** *Given a graph G, determine a bijection $\varphi : V(G) \to V(G)$ such that G^φ is a canonical form of G.*

Next, we turn our attention to the computational complexity of the three problems defined above. In particular, we are interested in how these problems are related to each other. A fundamental result is that the graph isomorphism problem and the graph automorphism problem are polynomial time equivalent, i.e., one can be solved in polynomial time using an oracle for the other problem.

Lemma 4. *The graph isomorphism and graph automorphism problems are polynomial time equivalent.*

Quite trivially, the graph isomorphism problem can be solved in polynomial time using an oracle for canonical labeling: we just canonize each of the input graphs and check equivalence.

Lemma 5. *The graph isomorphism problem polynomial time reduces to canonical labeling.*

On the other hand, it is an open problem whether canonical labeling reduces to isomorphism [13].

Interestingly, the isomorphism problem of more general objects polynomial time reduces to graph isomorphism. In turn, this also holds for the corresponding automorphism problems. First, we note that vertex-colored graph isomorphism is polynomial-time equivalent to graph isomorphism (follows from [80]). This can be achieved using a gadget construction, where we model the colors as small gadgets in the graph. A much more crucial result is that we can efficiently model arbitrary finite relational structures as graphs [80], and, in turn, solve the isomorphism problem on graphs instead of relational structures.

Lemma 6. *The isomorphism problem of finite relational structures polynomial-time reduces to graph isomorphism.*

This paves the way for using graph isomorphism, graph automorphism, or graph canonical labeling algorithms in a variety of different contexts, as discussed in Chapter 1. We remark that finite relational structures include the notions of directed graphs, graphs with self-loops, hypergraphs, and more.

2.1.3 Permutation Groups and Schreier-Sims

The automorphism group of a graph is a permutation group. Naturally, we require some tools to deal with permutation groups. We begin with basic definitions for permutation groups, followed by some fundamentals of the Schreier-Sims algorithm.

Permutation Groups. The *symmetric group* $\mathrm{Sym}(\Omega)$ is the set consisting of all permutations of the set Ω. A *permutation group* on a *domain* Ω is a group Γ that is a subgroup of $\mathrm{Sym}(\Omega)$, denoted as $\Gamma \leq \mathrm{Sym}(\Omega)$. Unless stated otherwise, we assume our permutation groups $\Gamma \leq \mathrm{Sym}(\Omega)$ are defined on the same domain $\Omega = \{0, \ldots, n-1\}$ as our graphs. In particular, we write $S_n \coloneqq \mathrm{Sym}(\{0, \ldots, n-1\})$. Given a set $S \subseteq \mathrm{Sym}(\Omega)$, we write $\langle S \rangle$ for the group *generated* by the elements of S, i.e., all elements that can be written as a product of elements of S. If $\langle S \rangle = \Gamma$ holds, we call S a *generating set* of Γ.

For $\varphi \in \Gamma$ and $\omega \in \Omega$, we write $\omega^\Gamma = \{\omega^\varphi \colon \varphi \in \Gamma\}$ for the *orbit* of ω under Γ. In other words, ω^Γ contains all the points in Ω that can be reached from ω by applying permutations of Γ. The partition of Ω into the orbits of Γ is called the *orbit partition*.

For a given subset of the domain $\Omega' \subseteq \Omega$, we define the restriction of Γ to Ω' as

$$\Gamma|_{\Omega'} \coloneqq \{\varphi|_{\Omega'} \mid \varphi \in \Gamma\}.$$

This restriction is not necessarily a permutation group since the images need not be in Ω'.

The *setwise stabilizer* is the group

$$\Gamma_{\{\Omega'\}} \coloneqq \{\varphi \mid \varphi \in \Gamma \wedge \varphi(\Omega') = \Omega'\}.$$

The group $\Gamma_{\{\Omega'\}}$ contains all permutations of Γ which stabilize Ω' as a set.

Let $\omega \in \Omega$ be a point, then the *pointwise stabilizer* of ω in Γ consists of all the elements of Γ that map ω to itself. Formally, we define

$$\Gamma_{(\omega)} \coloneqq \{\varphi \in \Gamma \mid \varphi(\omega) = \omega\}.$$

(Observe that $\Gamma_{(\omega)}$ is indeed a permutation group.) For a sequence of points $\Omega' \coloneqq (\omega_1, \ldots, \omega_m) \in \Omega^m$ we recursively take the pointwise stabilizer of all the elements in the sequence:

$$\Gamma_{(\omega_1,\ldots,\omega_m)} \coloneqq \begin{cases} \Gamma & \text{if } m = 0 \\ (\Gamma_{(\omega_1,\ldots,\omega_{m-1})})_{(\omega_m)} & \text{otherwise.} \end{cases}$$

We recall the *orbit-stabilizer* theorem [104]:

Theorem 7. *Let* $\Gamma \leq \mathrm{Sym}(\Omega)$*, and* $\omega \in \Omega$*. Then,* $|\omega^\Gamma| = \frac{|\Gamma|}{|\Gamma_{(\omega)}|}$.

With $\mathrm{supp}(\varphi) \coloneqq \{\omega \mid \omega \in \Omega \wedge \varphi(\omega) \neq \omega\}$ we denote the *support* of a permutation, which contains all points of Ω that are not stabilized by φ. The *support of a group* $\Gamma \in \mathrm{Sym}(\Omega)$ is the union of all supports of permutations of Γ, i.e.,

$$\mathrm{supp}(\Gamma) \coloneqq \{\omega \mid \omega \in \Omega \wedge \exists \varphi \in \Gamma : \varphi(\omega) \neq \omega\}.$$

We use the *cycle notation* for a permutation $\varphi\colon \Omega \to \Omega$. For example, the permutation of $\{1, \ldots, 7\}$ given by

$$1 \mapsto 3,\ 2 \mapsto 7,\ 3 \mapsto 4,\ 4 \mapsto 1,\ 5 \mapsto 5,\ 6 \mapsto 6,\ 7 \mapsto 2$$

we write as $(1, 3, 4)(2, 7)$. Algorithmically, the cycle notation enables us to read and store a permutation φ in time $|\mathrm{supp}(\varphi)|$.

Sifting. The routine described in the following is a crucial subroutine of the so-called Schreier-Sims algorithm [106]. In particular, the routine provides us with a data structure to dynamically manage permutation groups. Our brief introduction follows the description of [104].

We call a sequence of points $B = (\beta_1, \ldots, \beta_m) \in \Omega^m$ a *base* relative to a group $\Gamma \leq \mathrm{Sym}(\Omega)$ if $\Gamma_{(B)} = \{\mathrm{id}\}$. This means that the pointwise stabilizer of B fixes all the permutations of Γ, i.e., the pointwise stabilizer is trivial. For a generating set $\langle S \rangle = \Gamma$ and a base $(\beta_1, \ldots, \beta_m)$ we define

$$S_i \coloneqq S \cap \Gamma_{(\beta_1,\ldots,\beta_i)}.$$

Algorithm 1: Sifting an element into a Schreier structure.

```
1  function Sift
       Input: ➢ generating set S
              ➢ transversal table T
              ➢ base points B
              ➢ element φ
      Output: ≺ whether S, B and T remained unchanged
2      // for each base point...
3      for ( i = 1; i ≤ |B|; i = i + 1 )
4          // calculate which transversal element is needed to fix b_i
5          b_i := φ(B_i);
6          t := (T_i)_{b_i};
7          // stop if corresponding transversal element does not exist
8          if t = ⊥ then break;
9          // stabilize b_i in φ using transversal
10         φ := φ · t^{-1};
11         // here, b_i^φ = b_i holds
12     // if φ ≠ id, extend Schreier structure
13     if φ ≠ id and i ≤ |B| then
14         S := S ∪ {φ};
15         b_i := φ(B_i);
16         (T_i)_{b_i} := φ;
17         return false;
18     // if φ = id and i = |B| + 1, sift was successful
19     return true;
```

Here, $S_i \subseteq S$ denotes those generators, which stabilize the first i points of the base B. In particular, we observe that each $\langle S_i \rangle$ fixes the i-th base point of B, i.e., for all $\varphi \in \langle S_i \rangle$ it is true that $\varphi(\beta_i) = \beta_i$.

We call S *strong* relative to the group Γ and the base $(\beta_1, \ldots, \beta_m)$ if $\langle S \rangle = G$ and $\langle S_i \rangle = \Gamma_{(\beta_1,\ldots,\beta_i)}$ holds for all $i \in \{1, \ldots, m\}$. When S is strong, then each S_i suffices to generate the pointwise stabilizer $\Gamma_{(\beta_1,\ldots,\beta_i)}$, which in turn means that S contains sufficiently many generators to generate all the pointwise stabilizers related to the base B.

Given a subgroup $\Delta \leq \Gamma$, a *transversal* of Δ in Γ is a subset $T \subseteq \Gamma$ which satisfies $|T \cap gH| = 1$ for every coset gH of H in Γ. We construct a *transversal table* for a given base B and generating set S, which contains a transversal for each subgroup $\langle S_i \rangle$ in $\langle S_{i-1} \rangle$. We write T_i for the transversal of $\langle S_i \rangle$.

In order to determine the cosets of S_i in $\langle S_{i-1} \rangle$, we need to find the possible images of β_i in $\langle S_{i-1} \rangle$. Elements of $\langle S_{i-1} \rangle$ under which β_i has the same image are in the same coset of $\langle S_i \rangle$. Thus, we can differentiate transversal elements T_i according to the image of β_i under them. We denote by $(T_i)_b$ the element in T_i mapping $\beta_i \mapsto b$ if it exists. We set $(T_i)_b = \perp$ if such an element does not exist.

The cosets correspond to the orbit of β_i in S_{i-1}. Given an element $\varphi \in S_{i-1}$, we need to determine the image of β_i under φ, in order to determine the coset in which it is contained. The representative of the coset is the element t in the transversal T_i, which maps β_i to $\varphi(\beta_i)$. In particular, note that by forming the product $\varphi \cdot t^{-1} \in S_i$ we obtain an element that fixes β_i.

Description of Algorithm 1. The algorithm describes a *sifting* procedure, which can be used to test membership in a given permutation group whenever a strong generating set S and corresponding base B are available. Otherwise, if S is not strong, the sifting procedure computes a non-trivial permutation. In the version of the algorithm described here, this permutation is added to the generating set to ensure that the sifted element is covered. Since we assume B to be a base of the group, we never need to extend the base for this purpose. If an element *sifts successfully*, i.e., the procedure returns *true*, we know that it is contained in $\langle S \rangle$. This is also the case if the generating set S was not a strong generating set. On the other hand, if the sifting is unsuccessful, i.e., the procedure returns *false*, then the element was not in the group, or the generating set was not strong. In any case, the Schreier structure is extended whenever sifting is unsuccessful.

The algorithm repeatedly multiplies transversal elements to the initial element. The operations preserve the property of whether the initially given element is in the group. Each operation modifies the element so that it is contained in the next respective pointwise stabilizer. If an element sifts successfully, the resulting element is the identity. This gives us a representation of the group element as a product of transversal elements, i.e.,

$$\varphi \cdot t_1^{-1} \cdot t_m^{-1} = \mathrm{id} \iff \varphi = t_m \cdot \ldots \cdot t_1.$$

We will loosely refer to base, transversal table, and generating set together as a *Schreier structure*. As more and more elements are sifted, such a structure captures the progress made towards constructing the group.

Implementation of Algorithm 1. While the implementation of the algorithm follows the given outline, a few important adjustments are made. In particular, the so-called *Schreier vectors* are used to efficiently store transversal elements in terms of a description of how they can be obtained from elements in the generating set. We refer to [104] for a thorough description of this technique. Furthermore, when extending the Schreier structure (Line 13), we extend it by all elements which can be reached using φ and existing generators of S_i. In turn, b_i in φ is fixed using itself, and sifting continues.

In this thesis, we will mostly be concerned with the correctness of the Schreier structure. It should be noted that the efficiency of sifting is crucial for our routines. However, we resort to high-level techniques to improve efficiency (see Chapter 6). Our implementation of sifting itself follows the implementation used in NAUTY and TRACES closely, with adjustments enabling the sparse storage of generators and transversals (see Section 2.2.3).

Correctness of Algorithm 1. We give a crucial result related to Lemma 4.3.1 in [104], which will be required for correctness arguments later on:

Lemma 8. *Let Γ be a group, B a base of Γ, S a set of permutations in Γ and φ a uniformly distributed element in Γ. If $\langle S \rangle \neq \Gamma$, the probability that φ does not successfully sift through the Schreier structure defined by B and S is at least $\frac{1}{2}$.*

Proof. If $\varphi \notin \langle S \rangle$ holds, by definition, φ can not sift successfully through the Schreier structure defined by S and B. Since $\langle S \rangle \neq \Gamma$ there is at least one element $\varphi' \in \Gamma$ such that $\langle S, \varphi' \rangle \neq \langle S \rangle$. However, since $|\langle S, \varphi' \rangle| \geq 2 \cdot |\langle S \rangle|$, a uniform random element of Γ is not contained in $\langle S \rangle$ with probability at least $\frac{1}{2}$. □

The previous results are also the foundation for the Schreier-Sims algorithm [104]. The Schreier-Sims algorithm can, among many other things,

1. test membership of a permutation φ in Γ,
2. determine the order $|\Gamma|$, and
3. compute pointwise-stabilizers for a given sequence of points,

in polynomial time. In particular, note that given a Schreier structure, we can determine the order of the represented group by multiplying the sizes of all the contained transversals. For a description of the algorithm, particularly if no uniform random elements are available, we refer to [104].

2.2 Algorithms and Data Structures

We now discuss rudimentary data structures and algorithms that are used in practical graph isomorphism solvers. This will also enable us to state our algorithms more briefly and concisely. We mainly discuss particular operations on the data structures and how they can be implemented to satisfy certain theoretical runtime guarantees.

2.2.1 Sets, Lists, and Arrays

Let us describe a few basic data structures.

Small Integer Sets. First, we discuss how a set of integers $S \subseteq \{0, \ldots, n-1\}$ can be stored. We are interested in the following operations:

- Initialize the data structure as the empty set.
- Check for $i \in \{0, \ldots, n-1\}$ whether $i \in S$.
- Add $i \in \{0, \ldots, n-1\}$ to S, i.e., $i \in S$ holds after the operation.
- Iterate over all the elements contained in S.
- Reset S to the empty set.

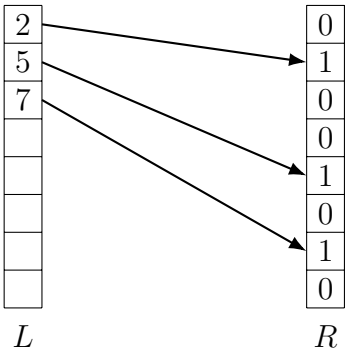


Figure 2.3: A small integer set representation for $S = \{2, 5, 7\}$.

The data structure consists of a Boolean array B of length n, and an additional list L that contains all the elements of the set. Initially, we set $B[i] = 0$ for all $i \in \{0, \ldots, n-1\}$ and initialize L as an empty list. Every time we *add* an element i, we check whether it is already in the set ($B[i] = 1$), and if not, set $B[i] = 1$ and add it to L. If we *reset* the set, we iterate through the list L and set $B[i] = 0$ for all $i \in L$. The data structure is illustrated in Figure 2.3.

We record the following basic facts about this data structure:

Lemma 9. *Given a set S as a small integer set data structure, the following operations can be implemented within the given worst-case runtime guarantees:*

- *Initialize the data structure in time $\mathcal{O}(n)$.*
- *Check for $i \in \{0, \ldots, n-1\}$ whether $i \in S$ in time $\mathcal{O}(1)$.*
- *Add $i \in \{0, \ldots, n-1\}$ to S in time $\mathcal{O}(1)$.*
- *Enumerate all elements of S in time $\mathcal{O}(|S|)$.*
- *Reset the data structure to the empty set in time $\mathcal{O}(|S|)$.*

Note that when starting from the empty set, resetting the data structure is amortized by the addition operations and therefore runs in amortized time of $\mathcal{O}(1)$.

We use this data structure in two different ways: either to store a set of small integers or to store a "list of unique small integers". Regarding the second use, note that we are also free to permute L, which means that, in principle, we are able to sort the list in place.

Reset Array. We also want to be able to store an arbitrary array of n integers, again with the ability to efficiently reset the array to an initial state.

The array is initialized with all 0 entries, and a *reset* operation is supposed to revert the array back to all 0. The idea we use is quite simple and related to the previous data structure: we additionally maintain a small integer set S of n integers. Every time an entry of the array is manipulated, we add the entry to S (unless it is already in S). On a reset, we iterate over all the elements in S to reset the array.

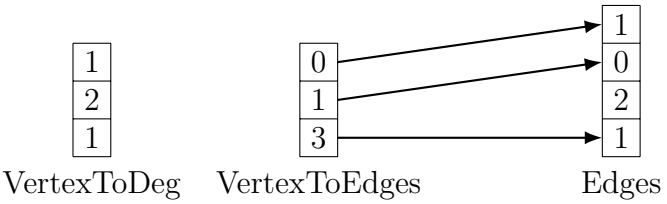


Figure 2.4: A sparse graph representation for a path of length 2, i.e., for the graph $G = \{\{0, 1, 2\}, \{(0, 1), (1, 0), (1, 2), (2, 1)\}\}$.

Lemma 10. *Given a reset array R, the following operations can be implemented within the given worst-case runtime guarantees:*

- *Initialization in time $\mathcal{O}(n)$.*
- *An entry $R[i]$ with $i \in \{0, \ldots, n-1\}$ can be accessed and changed in time $\mathcal{O}(1)$.*
- *A reset of the data structure can be performed in amortized time $\mathcal{O}(1)$.*

2.2.2 Sparse Graphs

While there are many ways to store graphs, we describe a specific data structure for *sparse graphs*. It makes use of adjacency lists. Indeed, it is precisely the sparse graph data structure employed by NAUTY and TRACES.

We store a graph $G = (V, E)$ using three arrays, called VertexToEdges, VertexToDeg, and Edges. VertexToEdges and VertexToDeg have length n, whereas Edges has a length of m. Each VertexToDeg[v] stores the degree for $v \in V$. Each VertexToEdges[v] stores a pointer into Edges. The neighbors of $v \in V$ are stored in

$$\text{Edges}[\text{VertexToEdges}[v]], \ldots, \text{Edges}[\text{VertexToEdges}[v] + \text{VertexToDeg}[v]].$$

An illustration of the data structure is depicted in Figure 2.4.

In our algorithms, unless stated otherwise, we assume that a given graph is precisely given as such a *sparse graph*. We record the following facts about the data structure:

Lemma 11. *Given a graph G as a sparse graph, the following operations can be implemented within the given worst case runtime guarantees:*

- *Given $v \in V(G)$, we can determine the degree $\deg(v)$ in time $\mathcal{O}(1)$.*
- *Given $v \in V(G)$, we can enumerate the open neighborhood $N(v)$ of v in time $\mathcal{O}(|N(v)|)$, and the closed neighborhood $N[v]$ in time $\mathcal{O}(|N[v]|)$.*
- *For two given vertices $v_1, v_2 \in V$, we can determine whether v_1 is adjacent to v_2 in time $\mathcal{O}(\min\{\deg(v_1), \deg(v_2)\})$.*

2.2.3 Sparse Symmetries

Since we are dealing with symmetries, or permutations in general, we also need a way to store them efficiently. In particular, we give three different data structures to represent permutations.

The first data structure stores a given permutation $\varphi \in S_n$ using an array Perm_φ where $\mathrm{Perm}_\varphi[i] = i^\varphi$. We refer to this as the *dense* data structure for storing a permutation. Clearly, this always takes up space $\Theta(n)$. Checking where a given point $i \in \{0, \dots, n-1\}$ is mapped to can be computed in time $\mathcal{O}(1)$ with a simple lookup operation. The support $\mathrm{supp}(\varphi)$ can be computed in time $\mathcal{O}(n)$ by iterating over all the points once and checking whether they map to themselves or not.

The second data structure stores φ as the *cycle notation* for φ. We consecutively store the cycles of φ, in order, in a list, using a unique separator element $\perp$ to denote the end of a particular cycle. This can clearly be achieved in space $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Checking where a given point i is mapped to, or computing the support of φ, is possible in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Hence, while this way of storing a permutation is space efficient, some computations take up more time.

The third data structure is more involved. We refer to it as the *dense-sparse* data structure. Essentially, this data structure combines the previous data structures. It enhances the dense data structure with a list that stores the support. In particular, given $\varphi \in S_n$, we store the permutation using two arrays Perm_φ and Supp_φ. The array Perm_φ is precisely the array from the dense structure, whereas Supp_φ stores $\mathrm{supp}(\varphi)$ as a list.

The dense-sparse structure is particularly useful in the following scheme. We initialize the structure once in time $\mathcal{O}(n)$ to the identity but then use it over and over again for multiple permutations. Note that if a permutation φ is given in cycle notation, and we have an initialized dense-sparse data structure, we can write φ to the dense-sparse structure in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Afterward, we can reset the data structure to the identity again in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. By iterating Supp_φ, the support can be computed in time $\mathcal{O}(|\operatorname{supp}(\varphi)|)$. Since we are also storing the dense representation, checking where φ maps a given point $i \in \{0, \dots, n-1\}$ can be computed in time $\mathcal{O}(1)$. Also note that given φ_1 and φ_2 as a dense-sparse data structure, we can easily compute $\varphi_1 \circ \varphi_2$ in time $\mathcal{O}(|\operatorname{supp}(\varphi_1)| + |\operatorname{supp}(\varphi_2)|)$.

2.2.4 Vertex Colorings

Let us now discuss a data structure for storing vertex colorings of a graph. Let π be a vertex coloring of a graph G. We store π in a specific manner, as described in the following. First, our colorings are *spacious*:

Definition 12. *A coloring $\pi : V(G) \to \{0, \dots, n-1\}$ is* spacious*, whenever for all $v \in V(G), \pi(v) = |\{v' \in V(G) \mid \pi(v') < \pi(v)\}|$ holds.*

This type of coloring is common among all state-of-the-art solvers and enables us to easily split up colors into smaller pieces, i.e., to make them finer. Unless stated otherwise, we consider vertex colorings to be spacious throughout this thesis. In particular, we also

assume that the vertex colorings given as the input to an algorithm are spacious. Of course, any vertex coloring can be made spacious by renaming the colors.

The data structure for vertex colorings consists of four integer arrays of length n, namely VertexToCol_π, ColToSize_π, Lab_π, and VertexToLab_π. We describe their meaning in the following:

- The array VertexToCol_π stores the color for each vertex, i.e., for all $v \in V(G)$ we maintain $\text{VertexToCol}_\pi[v] = \pi(v)$.

- The array ColToSize_π maps colors to their size, i.e., for all $c \in \pi(V(G))$ we maintain $\text{ColToSize}_\pi[c] = |\pi(c)|$, whereas all other entries may be arbitrary.

- The array Lab_π maintains a list of all vertices, ordered according to color. For each vertex $v \in V(G)$ there is an $i \in \{\pi(v), \ldots, \pi(v) + |\pi^{-1}(\pi(v))| - 1\}$ such that $\text{Lab}_\pi[i] = v$.

- The array VertexToLab_π maps vertices to their position in the array Lab_π, meaning for all $v \in V(G), \text{Lab}_\pi[\text{VertexToLab}_\pi[v]] = v$ holds.

We refer to the four arrays of storing π as our *default* data structure for a vertex coloring. Again, unless stated otherwise, we assume all vertex colorings considered throughout this thesis are provided using this data structure.

We record the following basic facts about the data structure:

Lemma 13. *Given a default data structure of a vertex coloring π of a graph G, the following operations can be implemented within the given worst-case runtime guarantees:*

1. *Given a vertex $v \in V(G)$, $\pi(v)$ can be computed in time $\mathcal{O}(1)$.*

2. *Given a color $c \in \pi(V(G))$, the size of the color $|\pi^{-1}(c)|$ can be computed in time $\mathcal{O}(1)$.*

3. *Given a color $c \in \pi(V(G))$, the vertices $\pi^{-1}(c)$ can be computed in time $\mathcal{O}(|\pi^{-1}(c)|)$.*

4. *The set of all (non-trivial) colors $\pi(V(G))$, can be computed in time $\mathcal{O}(|\pi(V(G))|)$.*

Proof. Claims (1) and (2) amount to simple array lookup of VertexToCol and ColToSize, respectively. For Claim (3), the respective elements can be found in

$$\text{Lab}_\pi[c], \ldots, \text{Lab}_\pi[c + \text{ColToSize}[c] - 1].$$

For Claim (4), we start at color 0. If we are at color c, the next color is c+ColToSize$[c]$. □

Algorithm 2: Checks if dense-sparse permutation is automorphism of graph.

```
1  function CheckAutomorphism
       Input: ➢ graph G = (V, E)
              ➢ coloring π
              ➢ dense-sparse permutation φ
      Output: ≺ whether (G, π) = (G^φ, π^φ) holds
   Auxiliary: ✏ small integer set M of length n
2    for ( each v in Supp_φ )
3      // set v' = v^φ
4      v' := Perm_φ[v];
5      // automorphism must preserve vertex coloring
6      if VertexToCol_π[v] ≠ VertexToCol_π[v'] then return false;
7      // automorphism must preserve neighborhood
8      x := 0;
9      for ( each n in N(v) )
10       add Perm_φ[n] to M;
11       x += 1;
12     for ( each n in N(v') )
13       if n ∉ M then
14         reset M;
15         return false
16       remove n from M;
17       x -= 1;
18     if x ≠ 0 then
19       reset M;
20       return false
21   return true
```

2.2.5 Testing Automorphisms

We now describe how to test whether a given permutation $\varphi : V(G) \to V(G)$ is indeed a symmetry of a given vertex-colored graph (G, π). Essentially, this means we test whether $(G, \pi) = (G, \pi)^\varphi$ holds. Such a procedure is commonly applied for every permutation a symmetry detection algorithm produces, in order to ensure correctness. Thus, it is indeed an important procedure that is continuously applied in practice. Also, it is a good excuse to get acquainted with the data structures described in this section.

Specifically, the algorithm works on our dense-sparse permutation data structure. We want to ensure it only runs in time that is proportional to the support and the edges incident to the support. The algorithm is given in Algorithm 2.

The algorithm assumes two additional properties. First, we assume that we have access to a small integer set M of length n, which is empty upon starting the algorithm. We make sure it is empty again upon terminating the algorithm. The idea is that we usually check many symmetries, and we only need to initialize the set M once for all of these

checks.

Secondly, we assume that φ is a bijection which is stored properly as a dense-sparse permutation. The actual implementation of the algorithm has additional checks in place to ensure that $\varphi|_{\mathrm{Supp}_\varphi}$ is indeed a bijection.

Description of Algorithm 2. The algorithm iterates over the support of φ. For each vertex v in the support, it then ensures that v and v^φ have the same color. Then, the following two for-loops ensure that the open neighborhoods of the two vertices are mapped onto each other by φ.

Correctness of Algorithm 2. We claim that under the assumptions stated above, the algorithm returns true if and only if φ is a symmetry of (G, π).

Let us make the following observations. For each $v \in \mathrm{supp}(\varphi)$, two properties are ensured:

1. The algorithm ensures $\pi(v) = \pi(v^\varphi)$, as checked in Line 6.
2. The algorithm ensures $\varphi(N(v)) = N(v^\varphi)$, as checked by the for-loops starting in Line 9 and Line 12.

Now if (1) and (2) are satisfied *for all* $v \in V(G)$, then it immediately follows that φ is a symmetry of the graph. We note that for each $v \notin \mathrm{supp}(\varphi)$, (1) is trivially satisfied because the vertex is just mapped to itself. However, (2) might indeed not be. Therefore, we need to prove that if $v \notin \mathrm{supp}(\varphi)$ and $\varphi(N(v)) \neq N(v^\varphi)$ hold, the algorithm correctly detects this.

If $\varphi(N(v)) \neq N(v^\varphi)$, then there is a $v' \in N(v)$ such that $v' \neq v'^\varphi$. Specifically, there is a v' for which $v' \in N(v)$ and $v'^\varphi \notin N(v)$ hold. Since v' is in $\mathrm{supp}(\varphi)$, the only case in which the algorithm does not detect this is if additionally $\varphi(N(v')) = N(v'^\varphi)$ holds (otherwise the algorithm would return *false* in the for-loop of Line 12, in the iteration for v').

Towards a contradiction, assume that $\varphi(N(v')) = N(v'^\varphi)$ holds. In this case, we know that both v' and v'^φ are neighbors of v. But this is an immediate contradiction to $v'^\varphi \notin N(v)$.

Runtime of Algorithm 2. The algorithm runs in time linear in the number of closed neighbors of vertices in the support of φ, i.e.,

$$\mathcal{O}(\sum_{v \in \mathrm{supp}(\varphi)} |N[v]|).$$

2.2.6 Efficient Orbit Algorithm

A second routine we want to briefly discuss is to compute the orbit partition of a group given as sparse generators. Again, efficiently keeping track of an orbit partition is an essential routine for symmetry detection algorithms.

Algorithm 3: Computes the orbit partition for a given generating set.

```
1  function ComputeOrbits
       Input: ➢ list of generators in cycle notation S = {φ_1, ..., φ_m}, where
              ⟨S⟩ ≤ S_n
       Output: ≺ union-find structure representing orbit partition of ⟨S⟩
2      initialize union-find data structure Δ of size n;
3      initialize dense-sparse permutation φ of size n;
4      for ( φ_i ∈ S )
5          load φ_i into dense-sparse permutation φ;
6          for ( each v in Supp_φ )
7              v' := Perm_φ[v];
8              union v and v' in Δ;
9          reset φ to identity;
10     return Δ
```

We make use of a union-find data structure. The structure is initialized to the discrete partition on n points. Recall that for k operations of any type, using path compression, the structure can be implemented such that it runs in a worst case running time of $\mathcal{O}(k\alpha(n))$, where α is the *inverse Ackermann function*.

Description of Algorithm 3. The algorithm begins by initializing a union-find data structure Δ. Then, the algorithm iterates over all the generators. For each generator, the respective partitions of two neighboring elements in each cycle are merged in Δ.

Runtime of Algorithm 3. Loading and iterating the cycles of the given generators can be implemented in the sum of the support of each of the generators, i.e.,

$$k \coloneqq \sum_{\varphi \in S} |\operatorname{supp}(\varphi)|.$$

We then perform k array lookups and union operations on a union-find data structure with domain size n. This gives a total runtime of $\mathcal{O}(\alpha(n)k)$.

2.3 Individualization-Refinement

We now describe the individualization-refinement paradigm. The exposition begins with the description of two important ingredients, namely the *refinement* and *cell selector*. Then, we describe the notion of *IR trees*. Lastly, we describe how IR trees can be used to solve our problems of interest. Our exposition of IR follows the one given in [76].

2.3.1 Refinement

In the following, we want to *individualize* vertices and *refine* vertex colorings. Individualizing vertices in a vertex coloring is a process that artificially forces a vertex into its

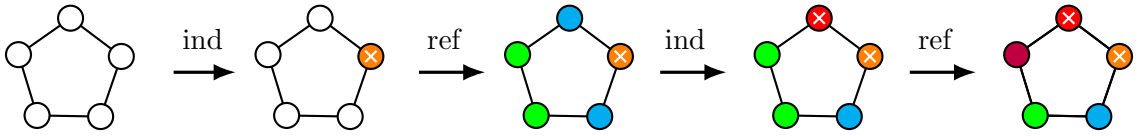


Figure 2.5: An illustration of individualizations and color refinement on a 5-cycle. Individualization steps split apart equally colored vertices (vertices marked with a cross are individualized). In turn, color refinement propagates this information.

own singleton color class. We use $\nu \in V^*$ to denote a sequence of vertices. (Here, * denotes the Kleene star.) In particular, we use ν to denote sequences of vertices which are individualized.

Next, we define the *refinements*. In IR, several different algorithms may be used as the refinement. Instead of describing a particular refinement, let us first define the necessary properties a refinement must satisfy for our purposes. A refinement is a function Ref which takes a graph G, a vertex coloring π of G, a sequence of vertices $\nu \in V^*$, and maps them to a coloring π'. Furthermore, a refinement must satisfy the following three properties:

1. It is invariant under isomorphism, i.e., $\mathrm{Ref}(G^\varphi, \pi^\varphi, \nu^\varphi) = \mathrm{Ref}(G, \pi, \nu)^\varphi$ holds for all $\varphi \in S_n$.

2. It produces finer colorings, i.e., $\mathrm{Ref}(G, \pi, \nu) \preceq \pi$ holds.

3. It respects vertices in ν as being individualized, i.e., $\{v\}$ is a singleton cell in $\mathrm{Ref}(G, \nu)$ for all $v \in \nu$.

We observe that if $\varphi \in \mathrm{Aut}(G, \pi)$,

$$\mathrm{Ref}(G, \pi, \nu^\varphi) = \mathrm{Ref}(G^\varphi, \pi^\varphi, \nu^\varphi) = \mathrm{Ref}(G, \pi, \nu)^\varphi$$

holds.

In practice, variants of the color refinement algorithm (also known as the 1-dimensional Weisfeiler-Leman algorithm) are commonly used as refinement. Intuitively, they classify vertices according to their degree, then the degrees of their neighbors, then the degrees of their neighbors' neighbors, and so on. Indeed, throughout this thesis, the only instantiation of Ref that we consider is color refinement.

We give a detailed account of color refinement in Chapter 4. Let us for now describe the *result* of color refinement, which is called an *equitable vertex coloring*. A coloring π is called *equitable* if for every pair of (not necessarily distinct) colors $i, j \in \{0, \dots, n-1\}$ the number of j-colored neighbors is the same for all i-colored vertices. For a graph (G, π), there is (up to renaming of colors) a *unique coarsest equitable coloring* π' finer than π. This is the coloring $\pi' = \mathrm{CRef}(G, \pi, \epsilon)$, where ϵ is the empty sequence. Considering individualization, $\mathrm{CRef}(G, \pi, \nu)$ is the unique coarsest equitable coloring finer than π in which every vertex in ν is a singleton with its artificial color. Note if vertices are

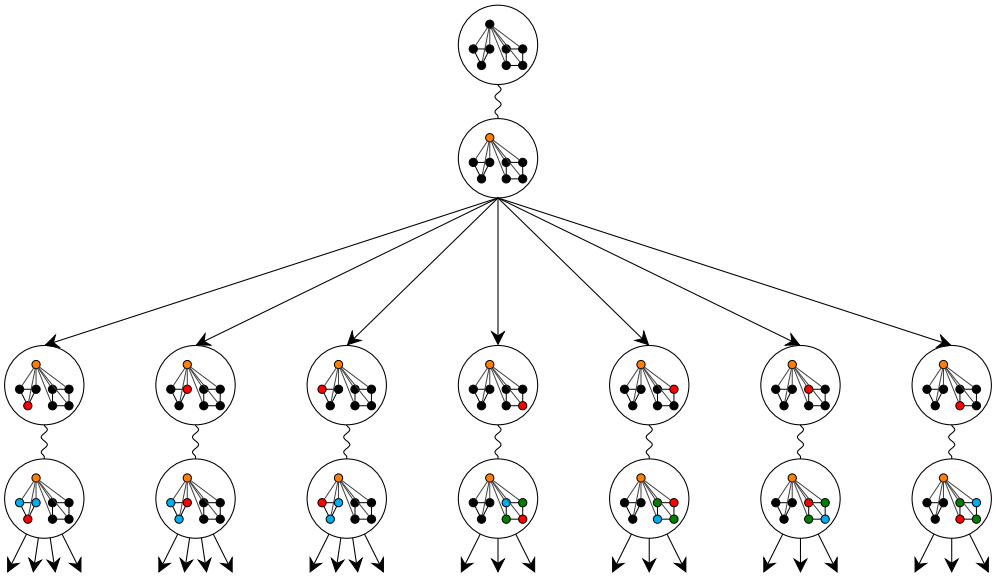


Figure 2.6: Construction of an IR tree. Pairs of nodes connected by a squiggly line correspond to one node in the IR tree. The upper node in a pair (other than the root) illustrates the individualization of a vertex, whereas the lower node in the pair shows the final coloring obtained after color refinement has been applied to the upper node. (The figure is adapted from [40].)

individualized, the artificial colors must be chosen isomorphism-invariantly, such that CRef respects the properties laid out above. Figure 2.5 illustrates the process.

We say two colored graphs (G_1, π_1) and (G_2, π_2) are *distinguishable (by color refinement)*, if with respect to the colorings $\mathrm{CRef}(G_1, \pi_1, \epsilon)$ and $\mathrm{CRef}(G_2, \pi_2, \epsilon)$

1. there is a color c with differently sized cells in G_1 and G_2, i.e.,
$$|\,\mathrm{CRef}(G_1, \pi_1, \epsilon)^{-1}(c)| \neq |\,\mathrm{CRef}(G_2, \pi_2, \epsilon)^{-1}(c)|,$$
2. or there are vertices $v_1 \in V(G_1)$, $v_2 \in V(G_2)$ of the same color, i.e.,
$$\mathrm{CRef}(G_1, \pi_1, \epsilon)(v_1) = \mathrm{CRef}(G_2, \pi_2, \epsilon)(v_2),$$
such that there is a color c within which v_1 and v_2 have a differing number of neighbors,
$$|\{(v_1, w) \in E(G_1) \mid \mathrm{CRef}(G_1, \pi_1, \epsilon)(w) = c\}| \neq$$
$$|\{(v_2, w) \in E(G_2) \mid \mathrm{CRef}(G_2, \pi_2, \epsilon)(w) = c\}|.$$

2.3.2 Selectors

If refinement classifies all vertices into different colors, determining automorphisms and isomorphisms for a graph is easy: this simply follows because colors have to be preserved. Otherwise, individualization is used to artificially single out a vertex inside a

non-singleton color class. The task of a *cell selector* is to isomorphism invariantly pick a non-singleton color class of the coloring. In the IR paradigm, all vertices of the selected color class are then individualized one after the other using some form of backtracking. After individualization, refinement is applied again, and the process continues recursively (see Figure 2.5). Formally, a cell selector is a function that takes a graph G and vertex coloring π of G, and maps them to a subset of vertices $V' \subseteq V(G)$ satisfying the following properties:

1. It is invariant under isomorphism, i.e., $\mathrm{Sel}(G^\varphi, \pi^\varphi) = \mathrm{Sel}(G, \pi)^\varphi$ holds for all $\varphi \in S_n$.
2. If π is discrete, then $\mathrm{Sel}(G, \pi) = \emptyset$ holds.
3. If π is not discrete then $|\mathrm{Sel}(G, \pi)| > 1$ and $\mathrm{Sel}(G, \pi)$ is a color class of π, i.e., there is a $i \in \{0, \ldots, n-1\}$ such that $\mathrm{Sel}(G, \pi) = \pi^{-1}(i)$ holds.

Cell selectors are analogous to determining the next variable to backtrack on in classical solvers of constraint programming, commonly referred to as the *variable order*.

2.3.3 IR Tree

With the functions Ref and Sel at hand, we are now ready to define the *IR search tree*. (We may also refer to these trees as *IR tree* or simply *search tree*.) For a vertex-colored graph (G, π) we use $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$ to denote the IR tree of (G, π) with respect to refinement operator Ref and cell selector Sel. We omit indices Sel and Ref if they are either apparent from context or of no particular importance. The search tree is constructed as follows: each node of the search tree corresponds to a sequence of vertices of G, the vertices individualized in that particular node of the tree.

1. The root of $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$ is the empty sequence ϵ.
2. If ν is a node in $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$ and $C = \mathrm{Sel}(G, \mathrm{Ref}(G, \pi, \nu))$, then its children are $\{\nu v \mid v \in C\}$, i.e., all extensions of ν by one vertex v of C.

We note that if $\mathrm{Ref}(G, \pi, \nu)$ is discrete, then $C = \emptyset$ and thus ν is a leaf of $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$. In other words, leaves of $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$ always correspond to discrete vertex colorings and, therefore, to permutations of V. With $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi, \nu)$ we denote the subtree of $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G, \pi)$ rooted in ν.

We recite the following crucial facts on isomorphism invariance of the search tree as given in [76], which follows directly from the isomorphism invariance of Sel and Ref:

Lemma 14. *For a vertex-colored graph (G, π) and $\varphi \in S_n$ we have $\mathrm{T}(G, \pi)^\varphi = \mathrm{T}(G^\varphi, \pi^\varphi)$.*

Based on this lemma, we can make the following observation regarding the symmetries of (G, π).

Corollary 15. *If ν is a node of $\mathrm{T}(G, \pi)$ and $\varphi \in \mathrm{Aut}(G, \pi)$, then ν^φ is a node of $\mathrm{T}(G, \pi)$ and $\mathrm{T}(G, \pi, \nu)^\varphi = \mathrm{T}(G, \pi, \nu^\varphi)$.*

Essentially, this tells us that symmetries are naturally represented in an IR tree. Let us make another observation. We observe that if C is the selected color class at a node $\nu \in \mathrm{T}(G,\pi)$, and $v \in C$, then $C \subseteq v^{\mathrm{Aut}(G,\pi)_{(\nu)}}$: Corollary 15 tells us that we can use elements of $\varphi \in \mathrm{Aut}(G,\pi)_{(\nu)}$ to obtain elements $\nu v^{\varphi} \in \mathrm{T}(G,\pi)$.

We have yet to mention how the search tree is used to find the automorphisms of a graph.

Lemma 16. *If ν and ν' are leaves of $\mathrm{T}(G,\pi)$, then there exists an automorphism $\varphi \in \mathrm{Aut}(G,\pi)$ such that $\nu = \varphi(\nu')$, if and only if $\mathrm{Ref}(G,\pi,\nu')^{-1} \cdot \mathrm{Ref}(G,\pi,\nu)$ is an automorphism of (G,π).*

Proof. Let $\varphi' = \mathrm{Ref}(G,\pi,\nu')^{-1} \cdot \mathrm{Ref}(G,\pi,\nu)$, which is a well-defined permutation on V since ν and ν' are leaves, which correspond to discrete colorings.

If φ is an automorphism with $\nu = \varphi(\nu')$, then $\mathrm{Ref}(G,\pi,\nu') \cdot \varphi = \mathrm{Ref}(G,\pi,\nu)$ holds, due to isomorphism-invariance of Ref. But then,

$$\varphi = \mathrm{Ref}(G,\nu')^{-1} \cdot \mathrm{Ref}(G,\nu') \cdot \varphi = \mathrm{Ref}(G,\nu')^{-1} \cdot \mathrm{Ref}(G,\nu) = \varphi',$$

proving the first direction.

For the other direction, assume $\varphi' \in \mathrm{Aut}(G,\pi)$. We observe that $\nu' = \varphi'(\nu)$, showing the claim. □

Let us also record that the individualized vertices in a leaf of the search tree actually form a base of the respective automorphism group [76].

Lemma 17. *If ν is a leaf of $\mathrm{T}_{(\mathrm{Ref},\mathrm{Sel})}(G,\pi)$, then ν is a base of $\mathrm{Aut}(G,\pi)$.*

Proof. Consider $\varphi \in \mathrm{Aut}(G,\pi)$ with $\varphi(v) = v$ for all $v \in \nu$. Now assume towards a contradiction that there is a $v' \in V$ such that $\varphi(v') \neq v'$. By applying φ, we thus get a different node in the search tree. By Lemma 14 and isomorphism invariance of Ref, we do however get

$$\varphi = \mathrm{Ref}(G,\pi,\nu)^{\varphi} \cdot \mathrm{Ref}(G,\pi,\nu)^{-1} = \mathrm{Ref}(G,\pi,\nu) \cdot \mathrm{Ref}(G,\pi,\nu)^{-1} = \mathrm{id},$$

which is a contradiction to the assumption that $\varphi \neq \mathrm{id}$. □

Let us further make the following observation.

Lemma 18. *A leaf τ can be mapped to exactly $|\mathrm{Aut}(G,\pi)|$ many leaves in $\mathrm{T}(G,\pi)$ using elements of the automorphism group $\mathrm{Aut}(G,\pi)$.*

Proof. From Lemma 17 we know that τ is a base of $\mathrm{Aut}(G,\pi)$. Now consider an element $\varphi \in \mathrm{Aut}(G,\pi)$. Clearly, τ^{φ} also corresponds to a leaf in the tree (Lemma 15) and τ^{φ} is a base as well. Now consider a different element $\varphi' \in \mathrm{Aut}(G,\pi)$, i.e., $\varphi' \neq \varphi$. Clearly, $\tau^{\varphi} \neq \tau^{\varphi'}$ holds since τ is a base. □

We call two leaves *equivalent* if there is an automorphism mapping one to the other. We refer to a leaf τ' as an *occurrence* of another leaf τ, if τ and τ' are equivalent.

2.3.4 Pruning with Invariants and Automorphisms

Let us fix a single leaf τ of the search tree. We may now search for automorphisms by comparing other leaves to τ. In the following, we call such a fixed leaf τ the *target leaf*.

Corollary 15, Lemma 17, and Lemma 16 tell us how to obtain all automorphisms of (G, π) by comparing other leaves to τ: first, for all automorphisms $\varphi \in \mathrm{Aut}(G, \pi)$, it follows that τ^φ is a node of the search tree (Corollary 15). In particular, since τ is a base of the automorphism group (Lemma 17), if $\varphi \neq \mathrm{id}$, $\tau \neq \tau^\varphi$ follows. In particular, we obtain φ by inspecting the corresponding colorings of τ and τ^φ (Lemma 16).

The search tree can be of exponential size in the size of the input graph since there can be an exponential number of automorphisms in the size of a graph. It should, however, be noted that even search trees of asymmetric graphs can be exponentially large in the input [87]. Therefore, in any case, we want to prune the tree as much as possible. Towards this goal, let us define a *node invariant*. A node invariant Inv is a function that essentially maps nodes of the search tree to some totally ordered set I. Formally, Inv takes as input a graph G, vertex coloring π, and sequence of vertices $\nu \in V^*$, and maps them to the totally ordered set I. A node invariant must satisfy two further properties:

1. The invariant must be isomorphism invariant, i.e., we require that $\mathrm{Inv}(G, \pi, \nu_1) = \mathrm{Inv}(G^\varphi, \pi^\varphi, \nu_1^\varphi)$ holds for all $\varphi \in S_n$.

2. If $|\nu_1| = |\nu_2|$ and $\mathrm{Inv}(G, \pi, \nu_1) < \mathrm{Inv}(G, \pi, \nu_2)$, then for all leaves $\nu_1' \in \mathrm{T}(G, \pi, \nu_1)$ and $\nu_2' \in \mathrm{T}(G, \pi, \nu_2)$ we require $\mathrm{Inv}(G, \pi, \nu_1') < \mathrm{Inv}(G, \pi, \nu_2')$.

If we have some invariant Inv, we immediately get the following result:

Lemma 19. *Let ν, ν' be leaves of $\mathrm{T}(G, \pi)$. If there is an automorphism $\varphi \in \mathrm{Aut}(G, \pi)$ such that $\nu = \varphi(\nu')$, then $\mathrm{Inv}(G, \pi, \nu) = \mathrm{Inv}(G, \pi, \nu')$ holds.*

Proof. This follows from the equalities $\mathrm{Inv}(G, \pi, \nu') = \mathrm{Inv}(G^\varphi, \pi^\varphi, \nu'^\varphi) = \mathrm{Inv}(G, \pi, \nu)$. □

Hence, even if we remove all nodes of the tree whose invariant deviates from the corresponding node invariant on the same level on the path to the target leaf, we can still retrieve the entire automorphism group. This operation is called *pruning using invariants*. Formally, we define $\mathrm{Prune}_{\mathrm{Inv}}(\tau', \nu')$ as the operation that removes the subtree rooted in node ν' if $\mathrm{Inv}(G, \tau') \neq \mathrm{Inv}(G, \nu')$, where $|\tau'| = |\nu'|$ holds and τ' is the prefix of length $|\nu'|$ of the target leaf.

Let us describe a natural invariant used in practice. For an equitable coloring π of a graph G, the *quotient graph* $Q(G, \pi)$ captures the information of how many neighbors vertices from one cell have in another cell, as described in the following. Quotient graphs are complete directed graphs in which each vertex has a self-loop. They include vertex colors as well as edge colors. The vertex set of $Q(G, \pi)$ is the set of all colors of (G, π), i.e., $V(Q(G, \pi)) := \pi(V(G))$. The vertices are colored with the color of the cell they represent in G. We color the edge (c_1, c_2) with the number of neighbors a vertex of cell c_1 has in cell c_2 (possibly $c_1 = c_2$). Since π is equitable, all vertices of c_1 have the same number of neighbors in c_2.

A crucial fact is that vertex-colored graphs are *indistinguishable by color refinement* if and only if their quotient graphs on the coarsest equitable coloring are equal. We should also remark that quotient graphs are indeed *complete invariants*, yielding the following property.

Lemma 20. *Let ν, ν' be leaves of $T(G,\pi)$. There exists an automorphism $\varphi \in \mathrm{Aut}(G,\pi)$ with $\nu = \varphi(\nu')$ if and only if $Q(G, \mathrm{Ref}(G,\pi,\nu)) = Q(G, \mathrm{Ref}(G,\pi,\nu'))$.*

We may also view quotient graphs as a way to color IR trees themselves, i.e., where we color a node ν with $Q(G, \mathrm{Ref}(G,\pi,\nu))$.

The search tree can also be *pruned using automorphisms*. For simplicity, we prevent the removal of the initial target leaf. Assume we already have $\varphi \neq \mathrm{id}$ of $\mathrm{Aut}(G)$ available. For all nodes ν where ν^φ is *not* a prefix of the target leaf, we define $\mathrm{Prune}_{\mathrm{Aut}}(\nu, \nu^\varphi)$ as the operation which removes the subtree rooted at ν^φ from the search tree. Intuitively, applying $\mathrm{Prune}_{\mathrm{Aut}}$ can only cut away parts of the search tree generated by already available automorphisms.

We record correctness arguments for $\mathrm{Prune}_{\mathrm{Inv}}$ and $\mathrm{Prune}_{\mathrm{Aut}}$ in the following lemma:

Lemma 21. *Let τ be the target leaf of a search tree $\mathrm{T}(G,\pi)$. Suppose T_P is the tree after any sequence of operations of type $\mathrm{Prune}_{\mathrm{Inv}}$ or $\mathrm{Prune}_{\mathrm{Aut}}$ has been performed on $\mathrm{T}(G,\pi)$. Let $\varphi_1, \ldots, \varphi_k$ denote all the automorphisms used for the operations $\mathrm{Prune}_{\mathrm{Aut}}$. Then,*

$$\langle \{\varphi \mid \tau^\varphi \text{ is a leaf of } T_P\} \cup \{\varphi_1, \ldots, \varphi_k\} \rangle = \mathrm{Aut}(G,\pi)$$

holds.

Proof. Since all elements in the definition above are clearly in $\mathrm{Aut}(G,\pi)$, it suffices to argue that every $\varphi \in \mathrm{Aut}(G,\pi)$ is generated by the given set.

First note that neither the target leaf τ nor any node corresponding to a prefix of τ can be removed through the pruning operations, by definition. If τ^φ is an element of the remaining search tree, or $\varphi \in \langle \varphi_1, \ldots, \varphi_k \rangle$ holds, the statement is immediately true. Therefore, consider the case where τ^φ has been removed by an application of $\mathrm{Prune}_{\mathrm{Inv}}$ and $\mathrm{Prune}_{\mathrm{Aut}}$. $\mathrm{Prune}_{\mathrm{Inv}}$ can not remove the leaf or any of its prefixes since $\mathrm{Inv}(\tau^\varphi) = \mathrm{Inv}(\tau)$ holds. Therefore, τ^φ was removed through an application of $\mathrm{Prune}_{\mathrm{Aut}}(\tau'^{\varphi'}, \tau'^{\varphi})$ with $\tau' \leq \tau$, using φ_i for some $i \in \{1, \ldots, k\}$. Clearly, there is a leaf $\tau^{(\varphi\varphi_i^{-1})}$ in $\mathrm{T}(G, \pi, \tau'^{\varphi'})$ and $\varphi = \varphi\varphi_i^{-1}\varphi_i$. By iterating the argument for $\tau^{(\varphi\varphi_i^{-1})}$, and since there are only finitely many applications of $\mathrm{Prune}_{\mathrm{Aut}}$, we must end up in a remaining leaf eventually. This, in turn, proves the claim. □

Lastly, we recall the following observation [66].

Lemma 22. *Let T_P be the tree that resulted from applying $\mathrm{Prune}_{\mathrm{Aut}}$ exhaustively to $\mathrm{T}(G,\pi)$. Then, T_P has*

$$|L(T_P)| = \frac{|L(T(G,\pi))|}{|\mathrm{Aut}(G,\pi)|}$$

many leaves.

Proof. Note that a leaf in the search tree T_P is an equivalence class of leaves in the original search tree. Let l be a leaf of $T(G,\pi)$. From Lemma 18 it follows that there are $|\mathrm{Aut}(G,\pi)| - 1$ other occurrences of l equivalent to l under $\mathrm{Aut}(G,\pi)$. □

2.3.5 IR, Isomorphisms, and Canonical Labeling

While the main focus of this thesis is to compute the automorphism group of a graph, IR search trees can also be used to solve graph isomorphism and graph canonical labeling.

In Section 2.1.2, we discussed the close relationship between the graph isomorphism problem and computing the automorphism group of a graph. This is, in particular, also true within the IR framework. Given two input graphs (G_1, π_1) and (G_2, π_2), consider the IR search trees $\mathrm{T}(G_1, \pi_1)$ and $\mathrm{T}(G_2, \pi_2)$. Analogously to the automorphism search, we can obtain *all* isomorphisms between (G_1, π_1) and (G_2, π_2) from the leaves of the two search trees. If $\nu \in \mathrm{T}(G_1, \pi_1)$ and $(G_1, \pi_1)^\varphi = (G_2, \pi_2)$, then $\nu^\varphi \in \mathrm{T}(G_2, \pi_2)$ follows immediately. Note the strong similarity to the automorphism search discussed in the previous sections. Indeed, all search strategies discussed previously and further strategies discussed throughout this thesis naturally translate between graph isomorphism and graph automorphism search.

For graph canonical labeling, we are given a single graph (G, π) and want to compute a canonical representative from the isomorphism class of (G, π). In practice, this is achieved by searching for a *minimal leaf* in $T(G, \pi)$ according to a *complete* node invariant Inv (see [76] for a more detailed discussion). Indeed, we need to ensure that the same minimal leaf (up to isomorphism) is found for every graph with the same isomorphism type as (G, π). Intuitively, the requirements for canonical labeling are more strict than for the other two problems: we are now looking for a specific leaf, and not just *any* pair of equivalent leaves that represent isomorphisms or automorphisms. Indeed, some techniques discussed throughout this thesis are *not applicable* for graph canonical labeling. In a sense, this matches the current complexity landscape of these three problems: the problems of computing graph isomorphisms and graph automorphisms are polynomial time equivalent, whereas canonical labeling may be harder (see Section 2.1.2).

2.4 Existing Solvers and their Strategies

We now describe state-of-the-art practical graph isomorphism solvers. All of these solvers are based on the IR paradigm. The descriptions below are collected from the papers describing said tools, by analyzing the source code of the newest version of the tool, or through personal communication with the authors of the respective tool.

2.4.1 nauty

The solver NAUTY [74, 76] for canonical labeling and symmetry detection was first published by Brendan Mckay in 1977. Its main innovation at the time was the use of automorphism pruning, which is reflected in its name reading "**n**o **aut**omorphisms? **y**es". However, over the years, the tool received many new optimizations and features.

The tool is able to read both a sparse (described in Section 2.2.2) and a dense (adjacency matrices) graph format. Among many options, it has a configurable refinement routine as well as invariants that can be activated by the user. Using these options, an experienced user may tailor the tool towards a specific application. NAUTY uses a depth-first traversal

of the IR tree, and keeps the lexicographically least leaf for automorphism comparisons [76]. The solver features two cell selectors, one which simply chooses the first non-trivial color of the current vertex coloring, and the other chooses the first color which is joined in a non-homogeneous fashion to the most number of other colors [76]. A notable feature of NAUTY is the use of the random Schreier-Sims algorithm for more thorough automorphism pruning.

The color refinement implementation of NAUTY is, in some ways, rudimentary: the vertex colorings stored by NAUTY have no ability to efficiently determine in which color class a vertex is contained. This is particularly prohibitive when refining colorings of large, sparse graphs. As a mitigation strategy, after the first individualization is made, NAUTY computes the distance from the individualized vertex to all the other vertices and splits color classes according to this distance.

Due to its more compact data structures and highly efficient low-level programming, the tool excels on small graphs [76]. In particular, it should be noted that NAUTY can be configured to only allocate very little memory on the heap. The efficient handling of small graphs is essential for applications such as exhaustive graph generation [75].

2.4.2 saucy

The solver SAUCY for symmetry detection was first published in 2004 [27]. The aim of the tool was to compute the symmetries of sparse graphs that exhibit a lot of symmetries with small support. Therefore, the design is mostly composed of features to accommodate this goal.

The tool reads sparse graphs stored in a similar data structure as described in Section 2.2.2. The tool outputs symmetries in a dense-sparse structure as described in Section 2.2.3, making it the only tool, apart from DEJAVU, that enables the user to read symmetries in time that is linear in the size of the support of the symmetry.

Internally, SAUCY uses a depth-first traversal of the search tree. A crucial concept of the tool is the use of "matched ordered partition pairs" [28]. To make the terminology more consistent, we refer to this concept as *matched vertex colorings*. Crucially, this concept enables the early detection of sparse symmetries. Here, "early" means symmetries are already detected at the inner nodes of the IR tree instead of at the leaves. Essentially, matched vertex colorings enable the solver to efficiently detect when partial automorphisms can be extended with the identity. This enables automorphism pruning to be applied earlier and more efficiently. We describe the concept of matched vertex colorings in more detail in Section 4.4.5.

The refinement routine is optimized for sparse graphs. It facilitates the efficient reversal of refinements and checks whether two colorings are "matched". The same holds for the cell selector used: the tool attempts to choose cells that steer search to matched colorings more quickly [28]. In subsequent publications, methods of dynamically switching the cell selector and methods for "conflict learning" have been described [23]. However, these features seem to come with severe downsides and limited applicability. Indeed, the features have not made it into the version of SAUCY that is in use by practitioners today (e.g., as used in [34]).

2.4.3 bliss

The solver BLISS by Tommi Junttila and Petteri Kaski was first published in 2007 [58]. The tool can solve both symmetry detection and canonical labeling. At first, the tool was described as an efficient reimplementation of NAUTY. However, it also contains unique features, as explained below. Again, BLISS employs a depth-first traversal of the IR search tree.

The tool exploits non-uniform components of the quotient graph during its traversal of the tree, which enables it to potentially determine automorphisms and conflicts early. Moreover, the tool features a form of *backjumping* [59]. The observation used is that every "failure" encountered in the search tree must be isomorphic to a failure encountered in the subtree of the target leaf. If this is not the case, the tool can backtrack immediately. We use a similar technique later on when discussing our "trace deviation set" technique in Chapter 6.

2.4.4 Traces

The solver TRACES for canonical labeling and symmetry detection was first published by Adolfo Piperno in 2008 [93]. Over the past years, the tool has received many more techniques and optimizations, which enabled it to surpass most other solvers on many types of graphs [76].

A major aspect of TRACES is that it chooses to traverse the search tree differently. The main routine uses a breadth-first search as opposed to a depth-first search. Clearly, this would only lead to the discovery of automorphisms at the very end of the search. Therefore, TRACES strategically intersperses random walks of the search tree, "experimental paths", to find automorphisms. This is done by starting from a leaf in the tree computed so far and executing a random walk (or alternatively using a specific heuristic) towards a leaf. The detected leaf is stored and compared to the leaves of other experimental paths, enabling TRACES to detect automorphisms. TRACES continuously searches for and stores additional leaves when computing experimental paths. This enables TRACES to detect leaves on a lower level in the tree, which might otherwise be pruned through invariant pruning. If it detects such a beneficial leaf on the next level, a special algorithm is applied, whenever TRACES is used for symmetry detection (as opposed to canonical labeling). We describe the algorithm in the following.

When in symmetry detection mode, the tool has another special search strategy: If TRACES detects a leaf l at some level i during experimental path exploration, a special mode of breadth-first search is used. Assume l is located right below node ν at level $i-1$ of the search tree. First, all other leaves at level i below node ν are computed. Let L_ν be the set of those leaves right below ν. Then, for every other remaining node ν' at level $i-1$, only a single leaf l' has to be computed to prune ν': we argue that this is the case, depending on whether l' is an occurrence of some leaf in L_ν or not. If the leaf l is an occurrence of some leaf in L_ν, this determines an automorphism mapping ν and ν'. Using this automorphism, ν' can be immediately pruned. Otherwise, l' is not an occurrence of any leaf in L_ν. But this immediately implies that there can not be an automorphism

mapping ν to ν'. In turn, since it suffices to find all automorphisms with respect to a single leaf, ν' can be pruned. In any case, only a single leaf of ν' is required to prune ν'.

To summarize, if a leaf was uncovered at level i right below some node ν, this method enables TRACES to finish the automorphism search by:

1. Computing *all* leaves at level i below *one* node ν.

2. Computing *one* leaf at level i for *all* other nodes $\nu' \neq \nu$ at level $i - 1$.

While TRACES is only able to read a sparse graph format, its color refinement routine is optimized to deal with graphs of all kinds of densities. Indeed, it contains different "splitting routines" depending on the density of a graph and color class. We discuss this in detail in Chapter 4.

Crucially, during color refinement, the tool records a so-called *trace* invariant (hence the name "TRACES"). At each step of the routine, i.e., for each color class that is dealt with in color refinement, some isomorphism-invariant information is written to the trace. In subsequent branches of the search tree, this information is continuously checked. If the isomorphism invariant information ever differs, then color refinement can be terminated early: the branches can not lead to isomorphic leaves. More details can again be found in Section 4.4.2.

TRACES also uses the random Schreier-Sims algorithm to facilitate automorphism pruning. The algorithm is, however, not always applied: a heuristic is used to infer whether applying the Schreier-Sims algorithm is beneficial [76]. The implementation of the Schreier-Sims algorithm itself is the same as used by NAUTY.

Before the IR search tree is traversed, TRACES preprocesses vertices of degree $0, 1$, or $n - 1$ of the graph. Furthermore, the cell selector never chooses cells containing vertices of degree 2. Special code then handles colorings in which color classes with vertices of degree 2 are not necessarily discrete. There is also a separate version of TRACES which can handle edge-colored and directed graphs [94].

2.4.5 Other Algorithms

We briefly discuss other notable implementations that were not described above. It should be mentioned in our benchmarks (Chapter 7), we only compare to the implementations listed above, and not the ones described below.

Individualization-refinement was first introduced into the realm of graph isomorphism in a series of papers by Parris and Read [91]. Interestingly, they describe a breadth-first traversal of the individualization-refinement tree.

The solver CONAUTO [70, 69] has a mode for isolated graph isomorphism testing, as well as symmetry detection of graphs. Notably, cell selection is performed dynamically at each level of the IR search tree. On a technical level, the solver is limited to a dense graph representation.

The tool NISHE by Greg Tener [108] features an interesting cell selector, which is dynamically adapted upon the discovery of automorphisms. The strategy is carefully designed

such that it is even applicable for canonical labeling. This, in turn, leads to provable polynomial runtime on the so-called Miyazaki graphs [82]. In [108], schemes to dynamically adapt and improve invariant pruning are also discussed. Furthermore, NISHE features a rudimentary parallel implementation.

The SCREWBOX [66] is a solver for graph isomorphism, which makes use of a randomized termination criterion.

Chapter 3

Search Tree Traversal

Historically, state-of-the-art IR solvers followed a depth-first search approach to traverse the IR tree (see Section 2.4). The tool Traces infamously broke away from this principle, mainly performing breadth-first search that is combined with a random traversal of the IR tree. This traversal strategy is at the very heart of the underlying algorithm. With its remarkable performance on difficult combinatorial graphs, Traces revealed that the traversal strategy is arguably the most important design choice in IR algorithms. This immediately raises the question whether there are theoretical, structural reasons why this traversal strategy is favorable. Going one step further, we can ask for optimal traversal strategies.

Traversal Model. In this chapter, we define a particular search problem in trees with symmetry. This search problem allows us to strip away all the other design choices that are made in the creation of an IR algorithm, isolating the core issue of the *traversal strategy in the IR search tree*.

An input consists of two trees. A node in a tree must either be a leaf, or have at least two children. Let n denote the size of the smaller one of the two trees and N the size of the larger one. The cost of our algorithms is the number of nodes that are "explored" by an algorithm.

Within this model, we consider three settings:

1. The Monte Carlo setting, in which algorithms have access to randomness, and may exhibit a bounded error.

2. The Las Vegas setting, in which algorithms have access to randomness, but may *not* err.

3. The deterministic setting, in which algorithms do not have access to randomness, and may of course not err.

Bounds. Within our theoretical model, we give upper and lower bounds. We are concerned whether the bounds imply a *linear* or *sublinear* runtime in the size of a given tree. Keep in mind that these trees model IR search trees: we can easily achieve linear runtime by simply traversing the entire IR search tree using, for example, depth-first or breadth-first search. In practice, linear runtime in the size of an IR search tree might, of course, mean exponential runtime in the size of a given input graph. In turn, *sublinear* time means we do not have to traverse the IR search tree in its entirety.

Setting	Lower Bound	Lower Bound (d-adic)	Upper Bound
Monte Carlo	$\Omega(\sqrt{n})$	$\Omega(\sqrt{n})$	$\mathcal{O}(\log(n)\sqrt{n})$
Las Vegas	$\Omega(n)$	$\Omega(\sqrt{n})$	$\mathcal{O}(d\log(N)\sqrt{n})$
Deterministic	$\Omega(n)$	$\Omega(n)$	$\mathcal{O}(n)$

Table 3.1: This table summarizes lower and upper bounds for the isomorphism problem implied by the results of this chapter. Here, $n = \min\{n_1, n_2\}$ and $N = \max\{n_1, n_2\}$, where the sizes of the trees are n_1, n_2 and d gives the maximum degree of the two input trees. We state separate lower bounds for trees with bounded (d-adic) and unbounded degree.

Regarding *upper bounds*, we provide a simple randomized Monte Carlo algorithm with a worst-case runtime of $\mathcal{O}(\sqrt{n}\log n)$ in the number of explored nodes. For trees of bounded degree, we design a Las Vegas algorithm, which has a worst-case expected runtime of $\mathcal{O}(\sqrt{n}\log N)$.

These algorithms are accompanied by nearly matching *lower bounds*, which show that $\Omega(\sqrt{n})$ nodes need to be explored for randomized Monte Carlo algorithms even on bounded degree trees. For unbounded degree inputs, Las Vegas algorithms must visit $\Omega(n)$ nodes in expectation. For deterministic algorithms, we get a lower bound of $\Omega(n)$ even for inputs of bounded degree. Table 3.1 provides an overview of the particular lower and upper bounds in all settings.

Characterization of IR Trees. A natural follow-up question may of course be: *is the model used in these lower and upper bounds even reasonable?* For the upper bounds, we may simply refer to our implementation of the algorithm used in the Monte Carlo upper bound, as described in Chapter 6. The lower bound constructions will however work using particular worst-case families of trees. In turn, the question is whether these trees can actually occur in practice: whether there is a cell selector and graph that produces precisely the given tree. In order to prove that these worst-case examples are actually reasonable, in the last section of this chapter, we turn to the question of which kinds of trees can appear as IR trees.

We give a characterization of which trees are IR trees. More precisely, we characterize which trees are IR trees using *color refinement* under a free choice of cell selectors. The characterization is constructive, meaning for a given tree that satisfies the requirements, we give a graph and cell selector that results in the given IR tree. Crucially, the characterization proves that all families used to prove the lower bounds of Table 3.1 are in fact IR trees.

3.1 A Model for Search Tree Traversal

We now describe our abstract model for IR search tree traversal. In our model search problem, the input consists of one or two hidden trees of which certain information is to be discovered. We begin by explaining how these trees can be explored by the algorithms.

Then, we describe further properties of the model trees, followed by a definition of the model search problem.

3.1.1 Exploration Model

We consider rooted trees in which there is a priori no bound on the degree of the vertices. However, we require that

1. no vertex has exactly one child (i.e., the tree is *irreducible*),
2. each leaf of the tree is colored,
3. there is no node that has exactly two children of which exactly one is a leaf.

The first requirement is natural, considering that cell selectors are not allowed to produce a singleton cell (see Section 2.3). The second requirement models that in a leaf, the discrete coloring reveals the isomorphism type of the leaf: we can efficiently discover isomorphisms and automorphisms at the leaf of IR trees (see Lemma 16). The third requirement stems from the fact that if a vertex of a color class of size 2 is individualized, the other vertex of that color is in turn also singleton. We discuss this in more detail in Section 3.5.1. (In fact, it turns out there are a few more necessary conditions regarding leaf colors and non-trivial automorphisms of trees, which are however of no importance for now: all trees we consider in our lower bounds will be asymmetric.)

Our *exploration model* for the trees restricts access of algorithms to the trees themselves and how they can be explored. We think of new information as being provided by an oracle to the exploration algorithm.

During execution, a node of the tree is either *explored* or *unexplored*. Whenever a new node is explored, the algorithm learns the number of children of the node. In particular the algorithm knows whether the node is a leaf or not. Furthermore, in case the node is a leaf, it learns its color.

At the beginning of an execution, everything except the root is deemed unexplored. The algorithm can only ever access previously explored nodes. The degree (i.e., the number of children) of an explored node v is always known. To explore further nodes, the algorithm can explore a child of a previously explored node. Specifically, when k is the number of children of node v, the algorithm can request the i-th child of v with $i \in \{1, \dots, k\}$, which thereby becomes explored. For this, the input has an arbitrary but fixed ordering for the children of each vertex.

The *cost* of the exploration is measured in the number of oracle accesses, i.e., the number of nodes that are ever visited by the algorithm. In particular, there is no cost for traversing previously explored parts of the tree, or any other auxiliary computation.

Figure 3.1 illustrates such an exploration of a tree. Note that while the algorithm always knows the degree of explored nodes, it is essentially unable to choose a *specific* child to explore, since in another input the ordering of the children may in fact be different.

More formally, a black box search tree $T = (V, E, \pi)$ consists of a rooted tree with colored leaves, and for each node an ordering of the children. With π_T, we denote the

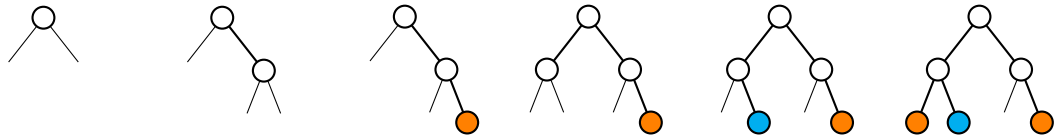


Figure 3.1: Example exploration in the black box search tree model. Starting from the root, the algorithm only ever knows explored nodes and their degrees. Through the use of an oracle, random children of explored nodes may then be queried.

coloring of a given black box search tree T. We omit the orderings from the notation. Of course all choices of orderings lead to proper search trees. The function $\pi\colon L(T) \to \mathbb{N}$ specifically only maps the *leaves of the tree* $L(T)$ to colors.

In our algorithms, we use the procedure

$$\texttt{NewChild}_T\colon V(T) \to V(T) \cup \{\bot\}$$

to explore the tree, which agrees with the previous description as follows. For an explored vertex v the algorithm chooses the smallest index of a previously unexplored child of v and queries the oracle for that child. If no unexplored child exists, the function returns $\bot$.

Moreover, in the description of randomized algorithms, we also use the function

$$\texttt{RandomChild}_T\colon V(T) \to V(T) \cup \{\bot\},$$

which returns a child chosen uniformly at random among all children of v, which means that it can in particular return previously explored children.

3.1.2 Isomorphism Invariance

So far we are lacking a crucial property of IR trees, namely the *isomorphism-invariance*. The core property of our trees is that the presence of leaves with equal colors implies the existence of symmetries of the trees. More specifically, they imply the existence of a color-preserving isomorphism. An isomorphism φ between two trees T_1 and T_2 is a bijection on vertices $\varphi\colon V(T_1) \to V(T_2)$, such that v is a child of v' if and only if v^φ is a child of v'^φ. In other words, φ is an isomorphism between the T_1 and T_2 interpreted as graphs, which additionally respects the root of the trees. A *color-preserving* isomorphism furthermore requires that $\pi_{T_1}(l) = \pi_{T_2}(l^\varphi)$ holds for all leaves $l \in L(V_1)$. This implies that leaves of a color can only be mapped to leaves of that same color. If $T_1 = T_2$, φ is an *automorphism*.

The crucial property that we require for all black box search trees is that whenever two leaves have the same color, we can derive an isomorphism:

Axiom (Complete Isomorphism Invariance)**.** *If $l_1 \in T_1, l_2 \in T_2$ and $\pi_{T_1}(l_1) = \pi_{T_2}(l_2)$, then there exists a color-preserving isomorphism $\varphi\colon V(T_1) \to V(T_2)$ such that $l_1^\varphi = l_2$.*

We should highlight that the axiom in particular has to hold for the case $T_1 = T_2$, yielding automorphisms (possibly the identity if $l_1 = l_2$).

The crucial consequence of the axiom is that it allows us to draw conclusions about the structure of unexplored parts of the search tree. For example, applying this knowledge enables us to conclude that the last remaining node of Figure 3.1 is blue. In the remaidner of this chapter, we assume that all input black box search trees, in particular if an input consists of *two* trees, adhere to this axiom. Also, all exploration algorithms operate in the exploration model defined above.

3.1.3 Isomorphism Exploration Problem

We are now ready to the state our problem of interest for black box search trees: the isomorphism exploration problem.

Problem 23 (Isomorphism Exploration)**.** *Given two search trees T_1 and T_2, compute leaves $l_1 \in T_1$ and $l_2 \in T_2$ with $\pi_{T_1}(l_1) = \pi_{T_2}(l_2)$, if they exist and return $\perp$ otherwise.*

For simplicity we will always assume that the trees are disjoint, that is $V(T_1) \cap V(T_2) = \varnothing$. This way we do not need to specify for oracle queries what tree they relate to. Furthermore, this allows us to indiscriminately refer to the colors of leaves, no matter if they are in T_1 and T_2. Essentially, we combine π_{T_1} and π_{T_2} into a unified coloring π.

We should remark that the problem as stated above of course relates to solving *graph isomorphism*, rather than computing graph automorphisms, using IR algorithms. However, as already pointed in Section 2.3.5, these two problems are strongly related in the context of IR algorithms. For example, consider the isomorphism exploration problem on *asymmetric* trees. Here, we can easily relate the isomorphism exploration problem to deciding whether a tree contains a non-trivial automorphism, as follows. We can combine two search trees into a single tree by adding a new root node, which is in turn connected to the root of the two given search trees. If we can decide whether there exists at least a single non-trivial automorphism of the resulting tree, we can solve the original isomorphism exploration problem. In any case, we believe that the isomorphism exploration problem seems more natural to work with.

3.2 Upper Bounds

We begin our exposition with upper bounds for the isomorphism exploration problem. All upper bounds are given in terms of algorithms solving the isomorphism exploration problem.

Let us begin with the trivial *deterministic* upper bound. We are given two input black box search trees T_1 and T_2. Our algorithm proceeds by exploring a single arbitrary node in each of the two input trees at a time. If we ever encounter two equally colored leaves, we have solved the problem. Once we finished exploring one of the trees, and no equally colored leaves have been found, we return $\perp$. Since we explore the trees one node at a time, the algorithm runs in time linear in the size of the smaller of the two input trees, i.e., $\mathcal{O}(\min\{n_1, n_2\})$.

Next, we discuss the Monte Carlo strategy, followed by the Las Vegas strategy for trees of bounde degree.

Algorithm 4: Random walk in a black box search tree.

```
1 function RandomWalk
      Input: ➢ black box search tree T
             ➢ explored vertex v ∈ V(T)
     Output: ≺ a random leaf of the search tree rooted at v
2   // continue to walk down in the tree, until leaf is reached
3   while deg(v) ≠ 1 do
4     v := RandomChild_T(v);
5   return v;
```

3.2.1 Monte Carlo Traversal

The central idea of the probabilistic isomorphism test discussed in this section is to repeatedly perform *random root to leaf walks* in the black box search trees.

Description of Algorithm 4. A random walk is performed by starting in the root node and repeatedly choosing uniformly at random a child of the current node, until a leaf is reached.

Cost of Algorithm 4. In the worst-case, the cost of the procedure is the height of the input tree T, i.e., $\mathcal{O}(h(T))$.

The probabilistic isomorphism test will exploit the following observation: assume we have two isomorphic trees T_1, T_2. Furthermore, assume we fix an arbitrary leaf $l \in T_1$ of the first tree. We call all leaves $l' \in L(T_1) \cup L(T_2)$ with $\pi(l) = \pi(l')$ *occurrences* of l. Naturally, if we find any occurence of l' in T_2, we have determined that the two trees are isomorphic. The algorithm tries to find occurrences of l solely through the use of random walks of the trees. Towards finding l, we always perform *two* random walks, one in T_1 and one in T_2. Since we assumed the trees *are isomorphic*, we are in fact *equally likely* to find an occurrence of l in T_1 or in T_2. But if the trees *are not isomorphic*, we can find occurrences of l *only* in T_1 (otherwise, due to the isomorphism invariance axiom, T_1 and T_2 would be isomorphic).

Description of Algorithm 5. Instead of using just a single leaf l, the algorithm uses two sets of leaves L_1 and L_2 for comparison. Whenever a new leaf is found, which is not an occurrence of a previously found leaf, it is added to the respective set of leaves and used for subsequent testing (see Line 21 and Line 22).

If a leaf is an occurrence of a previously found leaf, i.e., it is contained in L_1 or L_2, it either reveals an isomorphism between the trees or an automorphism (possibly the identity) of one of the trees (as checked in the for-loop starting from Line 16). This is again a consequence of the isomorphism invariance axiom. If the algorithm ever encounters two equally colored leaves, one in each of the two trees, it terminates (see Line 14 and Line 18). Otherwise, after a certain number of automorphisms have been found (the

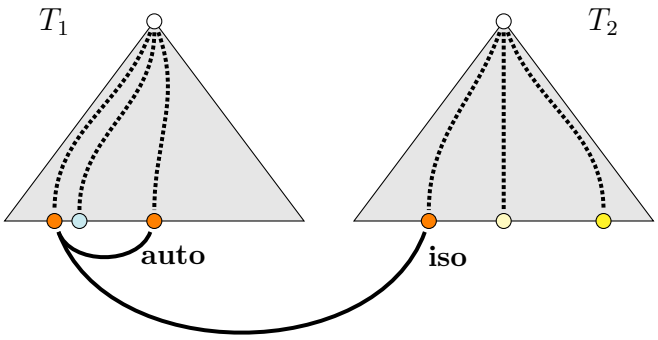


Figure 3.2: The probabilistic bidirectional search algorithm simultaneously samples leaves in both trees using random walks. It then tests for automorphisms within a tree and isomorphisms across trees to perform the probabilistic test.

number depends on the desired error bound), the algorithm concludes that the trees are *probably* non-isomorphic within the given error bound (see Line 24).

Again, if the trees are isomorphic, we find automorphisms and isomorphisms (that is *not* an automorphism) with equal probability. Hence, the idea is that we are highly unlikely to discover many automorphisms without also discovering an isomorphism (see further below for a formal proof). Figure 3.2 illustrates this key concept underlying the algorithm.

Correctness of Algorithm 5. The following lemma proves correctness of the algorithm.

Lemma 24. *Given black box search trees T_1, T_2 and a desired error probability ϵ, Algorithm 5 solves the isomorphism exploration problem with probability at least $1 - \epsilon$.*

Proof. First, observe that whenever a pair of leaves is returned, their color is checked for equality (Line 14 and Line 18). This ensures that if the algorithm returns a pair of leaves, the answer is always correct. The algorithm can therefore only fail to produce the correct output by not finding a suitable pair of equally colored leaves despite the fact that they exist. In particular, this implies that if the trees are non-isomorphic, the algorithm cannot err.

To bound the error probability, we view the computation as a sequence of *tests*. A test repeatedly performs random walks of the search trees until one automorphism (possibly the identity) or one isomorphism is found. Hence, each test can be described as a sequence of j iterations. In each iteration $j' < j$, neither l_1 nor l_2 produced an isomorphism or automorphism. During a test, the algorithm neither terminates, nor is c incremented. In iteration j of the test, an automorphism or isomorphism is found. Now, note that when T_1 and T_2 are isomorphic, leaves contained in $L_1 \cup L_2$ can equally likely be found in T_1 or T_2. Hence, finding an automorphism or isomorphism in a test is equally likely. In particular, the probability is $\frac{1}{2}$ for finding an isomorphism which is *not* an automorphism. Anytime we find an automorphism but no isomorphism, we increment c by 1. We terminate when c

Algorithm 5: Probabilistic bidirectional search in black box search trees.

```
1  function Isomorphism
       Input: ➢ black box search trees T_1, T_2
              ➢ probability ε
      Output: ≺ two leaves l_1 ∈ T_1, l_2 ∈ T_2 such that π(l_1) = π(l_2) with
              probability at least 1 − ε if such leaves exist, ⊥ otherwise
2      // initialize probabilistic abort criterion
3      c := 0;
4      e := ⌈−log_2(ε)⌉;
5      // initialize empty leaf storage
6      L_1 := L_2 := ∅;
7      // as long as probabilistic abort criterion not satisfied...
8      while c ≤ e do
9          // f_(aut,i) indicates automorphism found in T_i
10         f_(aut,1) := f_(aut,2) := false
11         // compute a random walk in each of the trees
12         l_1 := RandomWalk(T_1, root of T_1) ;
13         l_2 := RandomWalk(T_2, root of T_2) ;
14         if π(l_1) = π(l_2) then return (l_1, l_2);
15         // check color against all stored leaves
16         for ( i ∈ {1, 2} )
17             for ( l' ∈ L_(3−i) )
18                 if π(l_i) = π(l') then return (l_i, l') ;
19                 if π(l_(3−i)) = π(l') then f_(aut,(3−i)) := true ;
20         // add leaves to stored leaves
21         if ¬f_(aut,1) then L_1 := L_1 ∪ {l_1};
22         if ¬f_(aut,2) then L_2 := L_2 ∪ {l_2};
23         // manage probabilistic abort criterion if automorphisms
               found
24         if f_(aut,1) ∨ f_(aut,2) then c := c + 1;
25     // trees non-isomorphic with probability at least 1 − ε
26     return ⊥;
```

reaches e. Assuming the trees are isomorphic, the probability of this outcome is therefore bounded by $\frac{1}{2^e}$. □

Cost of Algorithm 5. We now analyze the worst-case runtime of Algorithm 5 in the cost model of black box search trees. This means that only Line 12 and Line 13 affect cost in the analysis. Since termination of the algorithm depends on random events, the running time of the algorithm is a random variable. Thus, we consider *expected* runtime.

We are now ready to analyze the running time of Algorithm 5.

Lemma 25. *Given black box search trees T_1, T_2 of heights h_1 and h_2, respectively, and error probability ϵ, Algorithm 5 has an expected worst-case runtime bounded by*

$$\mathcal{O}\left(\lceil \log_2\left(\frac{1}{\epsilon}\right)\rceil \cdot \max(h_1, h_2) \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}\right).$$

Proof. We calculate the expected number of leaves explored before termination. We may consider the number of leaves instead of nodes by adding the multiplicative factor $\max(h_1, h_2)$ for the maximum length of a root to leaf walk in the search trees to our runtime. (We explain subsequently how to improve this factor.)

We may assume that the input trees are non-isomorphic and thus that the algorithm terminates because the condition $c > e = \lceil -\log_2(\epsilon)\rceil$ was met. This suffices to give an upper bound since earlier termination due to the discovery of isomorphisms clearly only leads to a smaller expected running time.

Consider running $2\sqrt{|T_i|}$ iterations of the algorithm. We may assume that in the j-th iteration L_1 and L_2 each contain at least j leaves: otherwise, some previous iteration already discovered an automorphism or an isomorphism. Furthermore, we may assume that the probability to find a leaf is uniform across all leaves: if probabilities are non-uniform, the chance for finding some leaves repeatedly only increases (see [84]). The probability of finding an automorphism in L_i (with $i \in \{1, 2\}$) within j iterations is therefore at least $\frac{j}{|T_i|}$. After $\sqrt{|T_i|}$ iterations, the probability for finding an automorphism in T_i is then at least

$$\frac{\sqrt{|T_i|}}{|T_i|} = \frac{1}{\sqrt{|T_i|}}.$$

Hence, the probability of finding no automorphism after $2\sqrt{|T_i|}$ many steps is at most

$$\left(\frac{\sqrt{|T_i|} - 1}{\sqrt{|T_i|}}\right)^{\sqrt{|T_i|}} \leq \frac{1}{\mathbf{e}} < \frac{1}{2}.$$

We view the computation as a series of batches consisting of $2\sqrt{|T_i|}$ iterations each. For each of them, the probability for finding an automorphism is at least $1/2$. For termination, we need to find e many automorphisms. The expected number of batches is thus in $\mathcal{O}(e)$, which shows that the overall number of iterations is in $\mathcal{O}(e \cdot 2\sqrt{|T_i|})$. □

We can improve the bound on the runtime replacing the factor $\max\{h_1, h_2\}$ with the factor $\log_2(\min\{\sqrt{|T_1|}, \sqrt{|T_2|}\})$. To do so we alter the algorithm to take into account that the trees may be of very different sizes and also the trees may be quite unbalanced. To compensate for this we employ a doubling technique. However, we first need a bound for the expected length of the random root to leaf walks used in our algorithm.

Lemma 26. *In an n-node black box search tree the expected length of a random root to leaf walk (i.e., the running time of Algorithm 4) is in $\mathcal{O}(\log n)$.*

Proof. Let $g(T)$ be the expected length of a random root to leaf walk in tree T. Note that the number of leaves t of a black box search tree is in $\Theta(n)$ for an n-node tree. We will argue that among the trees T with t leaves and no inner nodes of degree 1, the value $g(T)$ is maximal if T is a binary tree in which all leaves are located on two consecutive levels. Since in such a tree even the maximum root to leaf distance is $\mathcal{O}(\log n)$, this proves the theorem.

First let T be a tree which has a vertex v with more than two children $u_1, \ldots, u_j$. Let T_i be the subtree of T rooted at u_i and assume without loss of generality that $g(T_i) < g(T_j)$ for $i < j$. Alter the tree T into a new tree T' by inserting a new node w as a child of v and then relocating the trees T_1 and T_2 so that their roots are now children of w instead of v. Note that while we are adding an inner node to the tree, the number of leaves remains unchanged. Then, conditional on the event that the random walk reaches v the expected length of the walk has increased. Thus $g(T') > g(T)$. Since there are only finitely many trees with t leaves, by induction it suffices now to consider binary trees.

Let T be a binary tree and suppose there are leaves ℓ_1 and ℓ_2 whose height differs by more than 1. Say ℓ_1 is on the level furthest from the root. There must be another leaf ℓ_3 whose parent p is also the parent of ℓ_1. Alter the tree to obtain a new tree T' by assigning ℓ_2 as the new parent of ℓ_3 and ℓ_2. Note that the transformation turns p into a leaf, and l_2 into an inner node. Hence, the number of leaves remains unchanged. Furthermore, since the height of ℓ_1 and ℓ_2 differs by more than 1, the node p is indeed further away from the root than ℓ_2. Thus, the tree being binary, the probability of a random walk reaching ℓ_2 is larger than that of reaching p. Therefore $g(T') > g(T)$. By induction this proves the theorem. □

This in turn suffices to prove the final theorem of this section.

Theorem 27. *There is an algorithm that solves the isomorphism exploration problem with probability at least $1 - \epsilon$ and expected worst-case runtime bounded by*

$$\mathcal{O}\left(\lceil \log_2\left(\frac{1}{\epsilon}\right)\rceil \cdot \log_2(\min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}) \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}\right).$$

Proof. Set $n = \min\{|T_1|, |T_2|\}$. For an integer s, we run the algorithm with a budget $2s$ that limits the number of walks that can be performed in each tree to s. Furthermore, we limit the length of the random walks by $h = c \log_2(s)$ for some suitable constant determined later. Whenever a random walk exceeds the length h, we abort the walk and ignore it. If the algorithm does not terminate within the alloted budget then we double s and restart. This guarantees that the number of queries does not exceed $\mathcal{O}(s \log s)$ when we run it with integer s.

At least in the smaller of the two trees, automorphisms are found with high probability whenever s exceeds $\sqrt{n}$. Indeed, by Lemma 26 the average length of a random walk in the smaller tree is in $\mathcal{O}(\log n) = \mathcal{O}(\log \sqrt{n})$. Thus, by Markov's bound with probability $1/2$, the random walks end in a leaf of height at most $\mathcal{O}(\log n)$. Thus, by the Chernoff bound, for sufficiently large s, with probability $1/2$ at least $1/4$ of the random walks end in a leaf of height at most $\mathcal{O}(\log n)$. We choose c so that this height is at most $c \log_2(n)$.

In case the graphs are isomorphic, automorphisms and isomorphisms are still found with equal probability. Thus our arguments for the probabilities remain in place since we essentially perform the same algorithm in pruned subtrees. Regarding the running time, note that the probability that the algorithm does not terminate with budget s decreases exponentially with s. That is, the probability is in $\mathcal{O}(a^{s/\min\{\sqrt{|T_1|},\sqrt{|T_2|}\}})$ for some constant $a < 1$ once $s > 2\min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. □

We should remark that the collision problem was previously exploited in the context of the group isomorphism problem [98]. However, in the group isomorphism problem additional structural information on the corresponding trees is known. Also, the idea of sampling with random walks was used for the isomorphism algorithm in [66], but that algorithm only uses a single leaf in the search tree and thus cannot achieve sublinear running time guarantees.

3.2.2 Las Vegas Traversal

The major drawback of the bidirectional search algorithm is that it makes errors. By considering trees of height 1, it is not difficult to see that a non-erring algorithm, even a randomized one, will need to query a linear fraction of the leaves to distinguish non-isomorphic trees. However, if the degree of the input graphs is restricted, we can beat this bound.

To do this, we basically strive to choose a specific set of nodes in both trees that ensures a "collision" of leaves. This guarantees that we find equally colored leaves, if they exist.

We refer to the maximum degree among the considered trees as d. The main new idea we use is to split the search tree in a *balanced* manner, followed by techniques to exploit isomorphism invariance. We want to note that the techniques for exploiting isomorphism invariance are inspired by the techniques described in [76, 107], in particular the special automorphism traversal of Traces (see Section 2.4.4), which essentially also use splits. However, rather than heuristically applying them, here, we perform them in a balanced and systematic way. Towards this goal we need the notion of a *split* (v, h), which is simply a node $v \in T_i$ at level h in one of the input trees. We define the *cost* of a split as a pair of numbers (s_1, s_2) as follows:

1. s_1 is the size of the tree T_{3-i} truncated at level h (i.e., the ball of radius h around the root).

2. If the tree T_i truncated at level h is non-isomorphic to the tree T_{3-i} truncated at level h, then $s_2 := s_1$, *otherwise* s_2 is the size of the subtree rooted in $v \in T_i$ at level h.

The intuition for our exploration strategy is that s_1 bounds the size of the subtree to be explored in T_{3-i}, while s_2 bounds the size of the subtree to be explored in T_i (up to logarithmic factors). While the definition for (2) may seem cumbersome at first, the idea is simply that if trees already differ in the first h levels, we can decide non-isomorphism by exploring all nodes in the subtree of T_i consisting of the first h levels and then at most as many vertices within the first h levels of T_{3-i}.

We call a split (v, h) a *balanced split* whenever its cost (s_1, s_2) satisfies

$$\max\{s_1, s_2\} \leq 4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}.$$

Note that slightly abusing terminology, in a balanced split the subtree with root v can be large, as long as the two trees truncated at level h are non-isomorphic and s_1 is sufficiently small.

At this point it might neither be clear how to find a balanced split nor that a balanced split always exists. However, assume for now that we are given a balanced split. In that case we can efficiently solve isomorphism (even deterministically) as follows. We perform breadth-first search up to level h in both trees T_1 and T_2, visiting all nodes $N_1 \subseteq V(T_1)$ and $N_2 \subseteq V(T_2)$ up to and including level h. We can conclude non-isomorphism immediately whenever the breadth-first search has finished level h and the two trees truncated at level h are non-isomorphic. We can thus assume now that these trees are isomorphic. By exploring all nodes up to level h (which is the level containing v), we surely explore the node v in one of the trees. Without loss of generality assume in the following that $v \in V(T_1)$.

In T_1, we explore *all* leaves L_v of the subtree rooted at *one* fixed node, namely v from the balanced split. Let $N_2' \subseteq N_2$ denote the set of nodes at level h in T_2. Then, we explore for each node v' in N_2' *one* arbitrary leaf $l_{v'}$ in the subtree rooted at v'. If the trees are isomorphic, there must exist some $v' \in N_2'$ that can be mapped to v with an isomorphism. Since we explored all leaves of v, the leaf $l_{v'}$ with ancestor v' must be isomorphic (equally colored) to one of the leaves in L_v. Figure 3.3 illustrates how the collision of leaves is enforced by this exploration strategy.

The procedure for exploring, within our model, the first h levels of the subtree rooted at a particular node v is described in Algorithm 6. Starting from a given node v, it performs breadth-first traversal until only leaves are left, or h levels have been explored. The algorithm is also given a cost limit s and the algorithm aborts if this limit is reached.

Using Algorithm 6 as a subroutine, Algorithm 7 gives an implementation in the exploration model of the entire algorithm just described.

Cost of Algorithm 7. Let us argue an upper bound for the runtime of Algorithm 7, assuming that we are given a balanced split. From the definition of a balanced split, we can conclude that $|L_v|$ and $|N_2|$ are bounded by $\mathcal{O}(d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\})$. Since exploration up to level h in T_1 (Line 4) may only explore as many nodes as exploration in T_2, we ensure that $|N_1| \leq |N_2|$ holds. Now, the last phase probes at most $\mathcal{O}(d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\})$ many paths, giving an overall upper bound of

$$\mathcal{O}\left(d \cdot h(T_2) \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}\right).$$

However, note that the factor $h(T_2)$ can be excessively large because the paths from level h to the leaves can be of length $\Theta(|T_2|)$. To prevent this, we alter the algorithm as follows. In T_2 we allocate a total budget of $c'\sqrt{|T_2|}\log(|T_2|)$ for some constant c' for all the level-h-to-leaves paths. By Lemma 26 the expected length of one such path is

Algorithm 6: Breadth-first search on a black box search tree.

```
1  function Subtree(v, h, s)
       Input: ➢ black box search tree T
              ➢ explored start node v ∈ V(T)
              ➢ height limit h
              ➢ cost limit s
      Output: ≺ (L, s') where L is the set of leaves of the subtree under v up to
              level h and s' is the number of explored nodes, or (⊥, ⊥) if cost
              limit did not suffice
2      // immediately terminate for trivial height limit
3      if h = 0 then return {v};
4      // initialize worklist N and empty set of leaves L
5      N := {(v, 0)} ;                        // (v, i) denotes node v at level i
6      L := {};
7      s' := 0;
8      // as long as the worklist is not empty...
9      while N ≠ ∅ do
10         // take an element v off the worklist
11         (v, h') := Some(N) ;                  // pick arbitrary element of N
12         N := N ∖ {(v, h')};
13         h' := h' + 1;
14         // now we iterate over all children of v
15         c := NewChild_T(v);
16         while c ≠ ⊥ do
17             s' := s' + 1 ;                          // keep track of cost limit
18             // reached cost limit?  terminate!
19             if s' > s then return (⊥, ⊥) ;
20             // reached height limit or node is a leaf?  add node to
                   result L
21             if h' = h ∨ deg(c) = 0 then L := L ∪ {c} ;
22             else
23                 // otherwise, add node to the worklist
24                 N := N ∪ {(c, h')}
25             c := NewChild_T(v);
26     // we stayed within the cost limit and return the result
27     return (L, s');
```

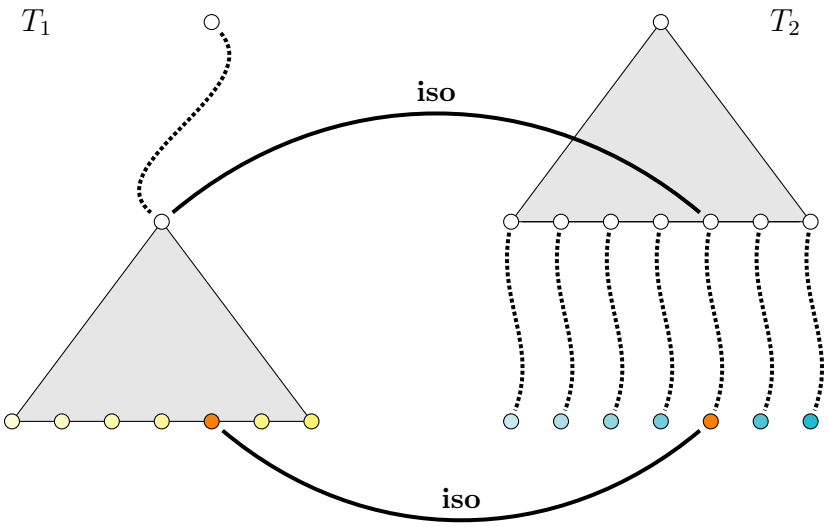


Figure 3.3: State of the search trees after termination of Algorithm 7. If trees are isomorphic, a node v_1 in T_1 at some level h must be mapped to some node v_2 in T_2 at level h by an isomorphism. But if that is the case, then a leaf below v_2 is isomorphic to some leaf below v_1.

Algorithm 7: Bidirectional search based on a given split.

```
1  function Isomorphism(T_1, T_2, v, h)
       Input: ➢ black box search trees T_1, T_2
              ➢ a split v, h with v ∈ V(T_1)
      Output: ≺ two leaves l_1 ∈ T_1, l_2 ∈ T_2 such that Col(l_1) = Col(l_2) if they
              exist, ⊥ otherwise
2      // explore the upper parts of trees
3      (N_2, s) := Subtree(root of T_2, h, ∞) ;
4      (N_1, _) := Subtree(root of T_1, h, s) ;                    // discovers v
5      // trivial case in which truncated trees not isomorphic
6      if N_1 = ⊥ or T_1 and T_2 up to level h non-isomorphic then
7          return ⊥;
8      // explore subtree under v
9      (L_v, _) := Subtree(T_1, v, ∞, ∞);
10     // ...and then a single leaf for each n ∈ N_2
11     for ( n ∈ N_2 )
12         l := RandomWalk(T_2, n) for ( l' ∈ L_v )
13             if π(l) = π(l') then return (l, l') ;
14     // no isomorphism found
15     return ⊥;
```

in $\mathcal{O}(\log |T_2|)$. Thus, by linearity of expectation, the expected total cost for the paths is $\mathcal{O}(\sqrt{|T_2|} \log |T_2|)$. By Markov's inequality for a suitable choice of c', with probability $1/2$ the total cost is in $\mathcal{O}(\sqrt{|T_2|} \log |T_2|)$. If the total cost exceeds this bound we simply restart the process.

Overall we can replace the factor $h(T_2)$ by $\mathcal{O}(\log(|T_2|))$, giving

$$\mathcal{O}(d \cdot \log_2(\max\{|T_1|, |T_2|\}) \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}).$$

More generally, it is easy to see that given a split of cost (s_1, s_2), the modification of Algorithm 7 runs in

$$\mathcal{O}(\log_2(\max\{|T_1|, |T_2|\}) \cdot \max\{s_1, s_2\}).$$

There is an interesting analogy to the runtime of the probabilistic bidirectional search algorithm. A main difference is that the runtime directly depends on the maximum degree of the trees.

The crucial question remains whether balanced splits always exist and whether they can be found efficiently. We first address the question of existence of balanced splits.

Lemma 28. *Let T_1, T_2 be black box search trees with maximum degree d. Then there exists a balanced split for search trees T_1 and T_2.*

In particular, if

- *h' is the maximal level for which the tree T_1 truncated at level h' is smaller than $4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$,*
- *the two subtrees up to level h' are isomorphic, and*
- *there are no leaves up to level h',*

then at least $\frac{3}{4}$ of the nodes at level h' in the smaller tree constitute balanced splits with cost $s_2 \leq 2 \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$.

Proof. We can assume w.l.o.g. that $|T_1| \leq |T_2|$. Let h' be the maximal level of T_2 where the size of the subtree up to level h' is smaller than or equal to $4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$.

If the subtrees up to level h' in T_1 and T_2 differ, we have found a balanced split. Furthermore, if there are leaves in the trees up to level h', we have found a balanced split as well. Hence, we assume that subtrees are isomorphic and no leaves are present.

We now argue that at least $\frac{3}{4}$ of the nodes at level h' in T_1 constitute balanced splits. Consider level h' of T_1 and T_2. Let $s_{h'} \leq 4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$ be the size of the subtree up to and including level h' in T_1. By assumption, the respective subtree of T_2 is of equal size. Furthermore, by assumption there are no leaves up to level h', implying that the tree contains at least $n_{h'} \geq \frac{1}{2} \cdot s_{h'}$ nodes at level h'.

Towards a contradiction, we assume $s_{h'} \leq 4 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. But then we can increment h': since $s_{h'} \leq 4 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$, it holds that $s_{h'+1} \leq 4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. This is a contradiction to the assumption that h' is maximal. Hence, we know $4 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\} < s_{h'} \leq 4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$.

We can immediately conclude $n_{h'} \geq 2 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. Naturally, there can be at most $\frac{1}{2} \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$ corresponding subtrees which have a size greater than $2 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$ with roots at level h' in T_1 (since $|T_1| \leq |T_2|$). Consequently, there must be at least $n_{h'} - \frac{1}{2} \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\} \geq \frac{3}{4} n_{h'}$ subtrees rooted at level h' with a size smaller than $2 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. This concludes the proof for both claims of the lemma. □

Thus, for all search trees there exist balanced splits. However, we still need to explain how to find balanced splits efficiently. As shown by the lower bound in the next section, it is impossible to do this deterministically in an adequate running time. We thus need a randomized procedure for finding balanced splits. We will show that the following method is suitable. Rather than pseudocode we give a high level description of the algorithm.

Algorithm 8 (Las Vegas Balanced Splits)**.**
Input: Black box search trees T_1 and T_2.
Output: The algorithm either returns a split (v, h) or determines that T_1 and T_2 are non-isomorphic.

1. Set cost limit $s \leftarrow 1$.
2. Perform breadth-first search in T_1 and T_2, limiting the size of the traversed subtree to s nodes (each). If after any level the breadth-first search trees for T_1 and T_2 differ, the algorithm terminates concluding non-isomorphism. Let h denote the level reached so far. If breadth-first search discovers a leaf v at or below level h, the algorithm terminates with the split (v, h).
3. For each $i \in \{1, 2\}$, uniformly and independently at random choose a node v_i at level h in both trees.
4. Compute breadth-first search starting from the node v_i in T_i, until one of the following conditions is met:
   - Breadth-first search finishes exploring the entire subtree of v_i, constituting the split (v_i, h).
   - Breadth-first search explored s nodes.

   This step is performed in parallel for both $i \in \{1, 2\}$ (i.e., each step alternates between the two), until one method succeeds in finding a split or both finish unsuccessfully. If at any point a split is found, the algorithm terminates immediately returning the split.
5. Set $s \leftarrow 2s$ and jump to Step 2.

It turns out that when the cost limit is sufficiently high the algorithm finds balanced splits with good probability. The intuition behind this is based on Lemma 28: once the algorithm reaches a sufficiently high level of the tree with breadth-first search, the majority of nodes constitute balanced splits.

From the description of the algorithm, the following corollary follows readily:

Corollary 29. *If Algorithm 8 chooses in some iteration an element v at some level h in Step 3 so that (v, h) constitutes a split with cost (s_1, s_2) and at this point $s \geq s_2$, then the algorithm terminates after this iteration and returns a split with cost (s'_1, s'_2) where $s'_1 = s_1, s'_2 \leq s_2$.*

Proof. By assumption, (v, h) constitutes a split with cost (s_1, s_2). This implies that the entire subtree below v is smaller than s_2. Consequently, since s is large enough, Algorithm 8 explores the entire subtree below v in Step 4, unless probing the other way in parallel terminates first: this only contradicts our claim if it results in a more expensive split. However, since a more costly split necessitates more steps to explore its respective subtree, running the search in parallel ensures that the cheaper split is found first. Note that $s_1 = s'_1$ holds for all nodes at level h, concluding the proof. □

Using this, we can prove that Algorithm 8 terminates with a balanced split with good probability, giving what we need for our isomorphism test:

Lemma 30. *If Algorithm 8 terminates with a split, it constitutes a balanced split with a probability of at least $\frac{3}{4}$.*

Proof. We can assume w.l.o.g. that $|T_1| \leq |T_2|$. Let h' be the maximal level of T_2 where the size of the subtree up to level h' is smaller than or equal to $4d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$.

First, we observe that we may always assume that the breadth-first trees explored in T_1 and T_2 up to level h' are isomorphic, since otherwise Algorithm 8 terminates immediately with no split (Step 2). Furthermore, since Algorithm 8 terminates when discovering leaves within the first h' levels in the breadth-first exploration (Step 2) and this result in balanced splits, we may assume that Algorithm 8 explores no leaves in the breadth-first search. We note that if Algorithm 8 finds no leaves, each doubling of s can only increase the level h reached by breadth-first search by at most 1.

Consider now Step 3 in the algorithm once level h' is reached. The algorithm picks a node of level h' uniformly at random. We now argue that with probability at least $\frac{3}{4}$ a node that is the root of a *small subtree* is chosen, i.e., a subtree that is smaller than $2 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$. This however follows readily from Lemma 28: since all subtrees at level h' are chosen for exploration with uniform probability, we can conclude that choosing a node that is the root of such a small subtree in T_1 has a probability of at least $\frac{3}{4}$. From the maximality of h' we can conclude that $s \geq 4 \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}$ (see proof of Lemma 28). Hence, Corollary 29 ensures that the algorithm terminates with a balanced split when choosing a node that is the root of a small subtree.

Furthermore, note that before level h' is reached, it is not possible for Algorithm 8 to return a split that is not a balanced split since the cost of probing is smaller than the bound for balanced splits. □

At level h' Algorithm 8 terminates with probability $\frac{3}{4}$. Careful inspection of the proof of Lemma 28 and Lemma 30 reveals that Algorithm 8 also terminates with probability at least $\frac{3}{4}$ after every consecutive doubling of s. While the cost (and therefore potential execution time) doubles, the probability of terminating before reaching the respective

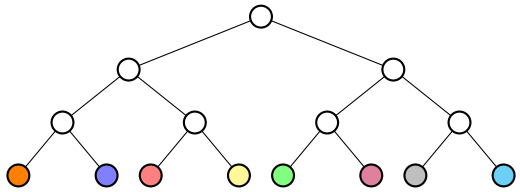


Figure 3.4: A search tree from the class $\mathcal{M}_3$.

cost quarters, which defines a geometric series: this results in an expected runtime of Algorithm 8 bounded by

$$\mathcal{O}(d \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}).$$

We now run Algorithm 8 and Algorithm 7 in series, which results in the desired algorithm: if Algorithm 8 terminates with non-isomorphism we are done. Otherwise, Algorithm 7 tests isomorphism with the provided split. We observe that whenever Algorithm 8 terminates with a split, the costs of the split are also bounded by s: the execution time can not be larger than the cost of the returned split. Running the previously described modification of Algorithm 7 with a split of cost s incurs expected cost bounded by $\mathcal{O}(\log_2(\max\{\sqrt{|T_1|}, \sqrt{|T_2|}\}) \cdot s)$. Using this, we can conclude the section with the following theorem.

Theorem 31. *Let T_1, T_2 be black box search trees with maximum degree d. There exists an algorithm for the isomorphism exploration problem with no error that has an expected worst-case runtime bounded by*

$$\mathcal{O}(d \cdot \log_2(\max\{\sqrt{|T_1|}, \sqrt{|T_2|}\}) \cdot \min\{\sqrt{|T_1|}, \sqrt{|T_2|}\}).$$

3.3 Lower Bounds

Now we prove lower bounds within the confines of the model of Section 3.1. Easy lower bounds can be obtained by considering input trees of height 1, where each leaf obtains a distinct color. However, we are interested in bounds that also apply to trees of bounded degree. We utilize the search tree family $\mathcal{M}_h$ for this purpose (see Figure 3.4). A tree is in $\mathcal{M}_h$, if it is a complete binary tree of height h such that leaves have pair-wise distinct colors, i.e., for all $(l_1, l_2) \in L(V(\mathcal{M}_h))^2$ with $l_1 \neq l_2$ it holds that $\pi(l_1) \neq \pi(l_2)$. With $\mathcal{M}$ we denote the smallest set which contains all the search tree families $\mathcal{M}_h$.

We remark that shrunken multipedes (see [86]) are graphs that produce search trees very similar to those in $\mathcal{M}_h$ when used as input for IR algorithms.

Generally, due to their uniformity, trees from $\mathcal{M}_h$ can only be distinguished or proven isomorphic by considering leaves. A traversal strategy must either conclude – with good probability $(\frac{1}{2})$– that the set of leaves of the trees are entirely disjoint or equal. In the case when trees are isomorphic, the traversal strategy must provide two leaves with equal colors.

3.3.1 Randomized Lower Bound

We prove lower bounds for the isomorphism problem for randomized algorithms that err. We use a particular type of exploration algorithm for our purposes. We call an algorithm *unadaptive* on a class of inputs if on each input from the class, the number of queries is always the same (in particular independent of randomness involved in the algorithm) and the queries performed by the algorithms on inputs from the class are independent of the answers given by the oracle. The queries may however still depend on the randomness involved in the algorithm. This means in particular that even when matching leaves have been found the algorithm will simply continue to run, possibly making further queries, and at some later point make a decision about the output.

Lemma 32. *If some (possibly randomized) algorithm A solves the isomorphism exploration problem with expected runtime $f(n)$ and error-probability ϵ then there is a randomized algorithm B with error-probability ϵ, which runs in time $\mathcal{O}(f(n))$ on $\mathcal{M}$ and for each $h \in \mathbb{N}$ is unadaptive on the class of inputs $\mathcal{M}_h$.*

Proof. If an algorithm A solving the problem with expected runtime $f(n)$ and error probability ϵ is given, then by repeating the algorithm and using Markov's inequality we can design an algorithm A' with a runtime bounded by $\mathcal{O}(f(n))$ (not just in expectation) that still has an error probability of ϵ. For this note even if the trees have been partially explored, it is possible to simulate the algorithm from scratch by pretending that explored nodes of the tree are unexplored.

To obtain the algorithm B we alter algorithm A' by simply pretending all discovered leaves have a randomly chosen previously unused color. More precisely, when a leaf is discovered, we pretend it has a color in $\{1, \dots, 2^h\}$ drawn independently and uniformly at random from the colors that have not been used yet (note that we can infer h whenever the algorithm reaches a leaf). We continue the simulation until A' halts. We then claim the input to be a yes instance if we found matching leaves and a no instance otherwise. This can only decrease the error probability in comparison to A'. □

The above lemma shows that every *adaptative* algorithm can be efficiently translated into an *unadaptive* algorithm. In turn, it suffices to prove a lower bound for unadaptive algorithms.

For our lower bound, we define a combinatorial problem of trees. Let M_h be the complete binary tree of height h, so that trees in $\mathcal{M}_h$ are colored versions of M_h. For two rooted trees U, S let $\mathrm{Inj}(U, S)$ be the set of root respecting injective homomorphisms from U to S. That is, the set contains the injective maps from $V(U)$ to $V(S)$ that map the root of U to the root of S and that map an edge of U to an edge of S.

From now on, fix a height h and consider the tree M_h. Let $\mathcal{U}_h$ be the set of trees U for which $\mathrm{Inj}(U, M_h)$ is non-empty. This set contains exactly the trees isomorphic to a subtree of M_h. For two trees $U_1, U_2 \in \mathcal{U}_h$ we let $P(h, U_1, U_2)$ be the probability that for uniformly chosen $\alpha_1 \in \mathrm{Inj}(U_1, M_h)$ and independently, uniformly chosen $\alpha_2 \in \mathrm{Inj}(U_2, M_h)$ the set $L(M_h) \cap \alpha_1(V(U_1)) \cap \alpha_2(V(U_2))$ is non-empty. For integers a, b define

$$P(h, a, b) = \max \{P(U_1, U_2) \mid |L(U_1)| = a \wedge |L(U_2)| = b\}.$$

Let $P(h,m) = \max\{P(h,a,b) \mid a+b \leq m\}$. We will argue that $P(h,m)$ constitutes an upper bound on the probability of success for a randomized algorithm for isomorphism exploration that queries at most m nodes.

Lemma 33. *Let B be an algorithm that is unadaptive on the class of inputs from $\mathcal{M}_h$. Suppose on inputs from $\mathcal{M}_h$ algorithm B makes m queries and has error probability ϵ. Then $1-\epsilon \leq P(h,m)$.*

Proof. Consider the behavior of the algorithm B on inputs from $\mathcal{M}_h$ with the colors $\{1,\ldots,2^h\}$ being randomly assigned bijectively to the leaves. The algorithm B explores subtrees T_1' and T_2', one in each of the input trees. Since the algorithm makes m queries, together, these trees can have at most m leaves. Our argument groups the possibilities in which B can query the oracle according to the topology of the two subtrees.

For two trees U_1 and U_2 consider the event E_{U_1,U_2} that T_1' is isomorphic to U_1 and T_2' is isomorphic to U_2. The event can of course only occur if $|U_1|+|U_2| \leq m$. Recall that algorithm B, being unadaptive, does not use the information on colors of the leaves provided by the oracle until the very end. Thus, the probability that B finds matching leaves on isomorphic inputs conditional to event E_{U_1,U_2} is $P(h,U_1,U_2)$.

We conclude that the probability that B finds matching leaves is at most $P(h,m)$. □

We remark that in our problem definition, the algorithm has to find two leaves of the same color. If the task only asked to decide whether the graphs are isomorphic, the algorithm could still guess, which would incur another factor of $1/2$.

We now show that the trees need to have sufficiently many leaves for $P(h,T_1,T_2)$ to be large.

Lemma 34. $P(h,a,b) \leq \frac{ab}{2^h}$.

Proof. For two trees T_1 and T_2 let $E(T_1,T_2)$ be the expected number of elements contained in the set $L(\mathcal{M}_h) \cap \alpha_1(V(T_1)) \cap \alpha_2(V(T_2))$, where α_1 and α_2 are taken independently and uniformly from $\mathrm{Inj}(T_1,\mathcal{M}_h)$ and $\mathrm{Inj}(T_2,\mathcal{M}_h)$, respectively. We define $E(h,a,b)$ in analogy to $P(h,a,b)$ as the maximum $E(T_1,T_2)$ over all choices of T_1 and T_2 with $|L(T_1)| = a$ and $|L(T_2)| = b$. By the Markov inequality it suffices to show that $E(h,a,b) \leq \frac{ab}{2^h}$.

Only vertices that are of distance h from the root in T_i can be mapped to a vertex in $L(\mathcal{M}_h)$. The automorphism group of $\mathcal{M}_h$ can map each leaf to every other leaf (i.e., acts transitively on the leaves). The graph $\mathcal{M}_h$ has 2^h leaves. Thus, for vertices $v_1 \in T_1$ and $v_2 \in T_2$ both of distance h from the root, the probability that $\alpha(v_1) = \alpha(v_2)$ is at most $\frac{1}{2^h}$.

By linearity of expectation the expected number of pairs (v_1,v_2) for which $\alpha(v_1) = \alpha(v_2) \in L(\mathcal{M}_h)$ is at most $\frac{1}{2^h} \cdot a \cdot b$. □

Theorem 35 (randomized lower bound)**.** *In the black box search tree model, a (possibly randomized making errors) traversal strategy runs in $\Omega(\min\{\sqrt{|T_1|}, \sqrt{|T_2|}\})$ worst-case cost for the isomorphism exploration problem, even on binary trees.*

Proof. By Lemma 32, it suffices to show the statement for an unadaptive algorithm B on $\mathcal{M}_h$. By Lemma 34, if B queries less than $\frac{1}{2}\sqrt{|\mathcal{M}_h|}$ nodes then both trees T_1 and T_2 uncovered by B have at most $\frac{1}{2}\sqrt{|\mathcal{M}_h|}$ leaves. But by the previous lemma we know that $P(h, \frac{1}{2}\sqrt{|\mathcal{M}_h|}, \frac{1}{2}\sqrt{|\mathcal{M}_h|}) \leq \frac{1}{4}$, which shows that the probability that B finds matching leaves in the two trees is at most $\frac{1}{4}$. This shows that B cannot find matching leaves with probability $\frac{1}{2}$. □

3.3.2 Deterministic Lower Bound

We exploit the randomized lower bound to obtain a strengthened deterministic one.

Theorem 36 (deterministic lower bound)**.** *In the black box search tree model, a deterministic traversal strategy runs in* $\Omega(\min\{|T_1|, |T_2|\})$ *worst-case cost for the isomorphism exploration problem, even on binary trees.*

Proof. Consider a deterministic algorithm on inputs from $\mathcal{M}_{2h}$, where $h = \log(n)$ and n is a power of 2. By Theorem 35 there are instances consisting of pairs of trees T_1, T_2 on which the algorithm makes $\Theta(\sqrt{2^{2\log(n)}}) = \Theta(n)$ queries in total. From the proof, we know that trees T_i can be chosen from the class $\mathcal{M}_{2h}$. For each $i \in \{1, 2\}$, we now remove from T_i all non-root vertices whose grandparents have not been explored by the algorithm (and thus who have not been explored either). More precisely, if v is at level h of T, we remove v whenever its ancestor on level $h-2$ has not been explored. Note that this guarantees that if v and v' have the same parent p, then v is a leaf if and only if v' is a leaf (so as to satisfy our requirement for black box search tree): this follows, because we remove the children of v if and only if we remove them from v', since all children have them same ancestor p.

Let T_i' be the resulting tree, respectively for each i. On the input pair (T_1', T_2') the algorithm behaves exactly the same as on (T_1, T_2) and thus also makes $\Theta(n)$ queries in total, however T_i' has at most $\mathcal{O}(n)$ vertices. This shows that on (T_1', T_2') the algorithm makes $\Omega(\min\{|T_1'|, |T_2'|\})$ queries. □

Note that balanced splits for the trees of $\mathcal{M}_h$ can be found almost trivially: after finding out the height h through a single walk, an arbitrary node at level $\frac{h}{2}$ will induce a balanced split. This shows that while $\mathcal{M}_h$ constitutes worst-case examples for probabilistic algorithms, this is not true for deterministic algorithms. And indeed, our deterministic lower bounds applies subtrees of trees in $\mathcal{M}_h$ which have leaves on different levels.

This concludes the proof of all the results stated in Figure 3.1.

3.4 Monte Carlo, Las Vegas, and Traces

Using our new insights we can explain why some of the strategies used by Traces turn out to be highly efficient. As discussed previously, Traces uses breadth-first search intertwined with random walks of the search tree (see Section 2.4.4). In particular, this is often done in a cost balancing manner, such that the number of random walks

is proportional to the cost of breadth-first search. This, in turn, often leads to the automorphism group being found in time proportional to the square root of the search tree size. For sophisticated pieces of software such as Traces, the traversal strategy is, of course, not the only deciding factor when it comes to running time. However, generally, the experimental paths often enable Traces to discover automorphisms much earlier than solvers solely utilizing deterministic depth-first traversal. Hence, automorphisms are available more quickly for pruning. Overall, in some sense, Traces emulates some of the techniques described in our Monte Carlo algorithm.

Moreover, Traces also sometimes uses some techniques of the Las Vegas algorithm we describe. Specifically, it performs splits in its "special traversal" strategy for automorphism groups (see Section 2.4.4). When Traces detects a leaf on level h with parent v, in our terminology it executes the split $(v, h-1)$. Since some graph classes in the benchmark suite (see Chapter 7) contain leaves at a height of 2 or 3 (e.g. the graph classes latin-sw, sts-sw, pp), Traces in practice turns out to frequently perform splits that are fairly balanced. This results in significant speedups over other solvers (e.g., see runtime of Traces on aforementioned classes in Section 7.2).

3.5 Characterization of IR Trees

The bounds of Figure 3.1 only apply to the search problem in arbitrary trees with symmetries, independent of whether they originate from actual IR computations or not. In this section, however, we turn to precisely characterizing which trees can occur as IR trees. We should make clear that throughout this section, IR trees will specifically mean IR trees using *color refinement* as a refinement, and *quotient graphs* as the invariant. In particular, the quotient graphs are interpreted as a *vertex coloring* of the IR trees. As discussed in Section 2.3, these choices reflect the choices used in practice. Overall, given a tree (T, π) satisfying the necessary conditions, we give a procedure that produces a cell selector and graph which lead to the IR tree (T, π), up to a renaming of the vertices and colors.

Arising from a branching process, all IR trees are rooted and all inner vertices naturally have at least 2 children. Such trees are called irreducible (or series reduced). It turns out that *not* all irreducible trees are IR trees.

Theorem 37. *An irreducible tree is an IR tree if and only if there is no node that has exactly two children, of which exactly one is a leaf.*

Beyond this, our characterization also fully describes how color classes may be distributed in a given tree (Section 3.5.1). It turns out that there are several simple restrictions, in particular for vertices that have precisely two children, but apart from that all colorings can be realized and, in particular, any number of symmetries can be ensured.

Overall, when modeling a graph that is supposed to produce a particular IR tree, two major difficulties arise, roughly summarized as follows:

1. The effect of color refinement on the graph needs to be kept under control.

2. The shape of the IR tree may dictate that symmetries must be simultaneously represented in distinct parts of the graph.

We resolve these issues using various gadget constructions specifically crafted for this purpose. We introduce *concealed edges*, which allow us to precisely control the point in time at which the IR process is able to see a certain set of edges and thus color refinement to take effect, resolving issue (1). By combining concealed edges with gadgets enforcing particular regular abelian automorphism groups we can synchronize symmetries across multiple branches of the tree, resolving issue (2).

Here, as the main tool we show the following. As an additional restriction, which stems from the structure of IR trees, we consider only trees where all leaves can be mapped to the same number of other leaves via symmetries (i.e., under automorphisms all *leaf orbits* have the same size). We show that each such tree T can be embedded into a graph H_T, such that H_T restricts the symmetries of T in a particular way. Intuitively, we keep just enough symmetries to allow leaves to be mapped to each other whenever this is possible in T. We thereby effectively *couple* leaf orbits so that when fixing one leaf, all other leaves are fixed as well. Formally, we call a group Γ *semiregular* on a set T, whenever for each pair $t_1 \in T, t_2 \in T$ there is at most one element of Γ which takes t_1 to t_2. We prove the following theorem.

Theorem 38. *Let T be a colored tree in which all leaf orbits have the same size. There exists a graph H_T containing T as an automorphism invariant induced subgraph so that the action of $\mathrm{Aut}(H_T)$ is faithful on T and semiregular on the set of leaves of T. Moreover, $\mathrm{Aut}(H_T)$ induces the same orbits on T as $\mathrm{Aut}(T)$.*

Again, we prove the theorem in a constructive manner. All steps can be easily converted into an algorithm that takes as input an admissible (i.e., compatible with our necessary conditions from Section 3.5.1) colored tree T and produces a graph and cell selector with IR tree T.

3.5.1 Necessary Conditions for IR Trees

Let us first formalize our *coloring* of IR trees. We call sequences (or t-tuples) of vertices $\nu_1 \in V(G_1, \pi_1)^t$ and $\nu_2 \in V(G_2, \pi_2)^t$ *distinguishable*, if the graphs $(G_1, \mathrm{CRef}(G_1, \pi_1, \nu_1))$ and $(G_2, \mathrm{CRef}(G_2, \pi_2, \nu_2))$ are distinguishable by color refinement. By extension, this means two nodes in the IR tree are distinguishable, if and only if the quotient graphs of their respective colorings are equal. In order to make this explicit, we color the nodes of the IR trees themselves using its corresponding quotient graph, i.e., we color a node ν with $Q(G, \mathrm{Ref}(G, \pi, \nu))$. Hence, distinguishing nodes of the tree using the quotient-graph invariant reduces to checking whether they have the same color.

We now collect the necessary conditions for the structure of IR trees. Since IR trees are the result of a branching process, they are naturally irreducible (no node has exactly one child).

Lemma 39. *IR trees are irreducible.*

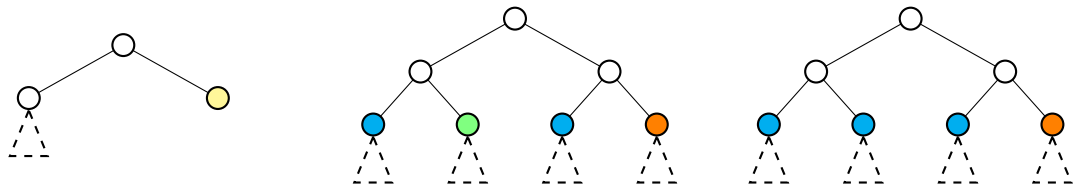


Figure 3.5: Forbidden structures in asymmetric binary IR trees.

Naturally, indistinguishable leaves can be mapped to each other using an automorphism, and this implies an automorphism of the tree.

Lemma 40. *Let l_1, l_2 be two leaves of an IR tree (T, π). If l_1 and l_2 are indistinguishable, there is an automorphism $\varphi \in \mathrm{Aut}(T, \pi)$ mapping l_1 to l_2.*

Proof. Follows directly from Lemma 15. □

The following condition also follows directly (see Lemma 18).

Lemma 41. *A leaf l can be mapped to exactly $|\mathrm{Aut}(G, \pi)|$ leaves in $\Gamma(G, \pi)$ using elements of the automorphism group $\mathrm{Aut}(G, \pi)$.*

This means all classes of indistinguishable leaves are equal in size.

Since we are using color refinement, equitable partitions and, hence, quotient graphs only ever become finer and more expressive. Therefore, the following holds.

Lemma 42. *Let n_1, n_2 be two nodes of an IR tree where n_i is on level l_i.*

1. *If $l_1 \neq l_2$, then n_1 and n_2 are distinguishable.*
2. *Consider the two walks starting in the root and ending in n_1 and in n_2, respectively. If in these walks two nodes on the same level are distinguishable, then n_1 and n_2 are distinguishable.*

Some further restrictions apply specifically in the case of cells of size 2.

Lemma 43 (Forbidden Binary Structures)**.**
1. *If a node n has two children n_1 and n_2, then it cannot be that exactly one of the children n_1 or n_2 is a leaf (see Figure 3.5, left).*
2. *If n_1, n_2 are any two nodes and n_1 has exactly 2 children then the multiset of colors of the children of n_1 and n_2 are equal or disjoint (Figure 3.5, middle and right).*

Proof. Part 1 follows from the fact that individualizing one vertex in a cell of size 2 also individualizes the other vertex of the cell.

For Part 2 we note that individualization of a child of n_1 also individualizes the other child of n_1 and vice versa. This implies that if a child c_2 of n_2 has the same color as some child c_1 of n_1, then by definition, individualization of c_1 and c_2, respectively, produces indistinguishable colorings. So, in this case, there is a one-to-one correspondence between the colors of the children of n_1 and those of n_2. □

It is easy to see that if at any point the cell selector chooses differently sized cells in different branches, the branches subsequently become distinguishable. However, if we assume cell selectors only base their decision on the quotient graph, this restriction applies earlier. More specifically, we call a cell selector *quotient-graph-based* whenever the result of the cell selector depends only on the quotient graph rather than other aspects of G and π, i.e., essentially, we have $\mathrm{Sel}(Q(G,\pi))$ instead of $\mathrm{Sel}(G,\pi)$. Then, the following holds.

Lemma 44. *If two nodes n and n' in an IR tree are indistinguishable, then their parents have the same number of children. If additionally the cell selector is quotient-graph-based, then n and n' also have the same number of children.*

Restricting the cell selector to quotient graphs thus changes whether we can distinguish nodes with a differing number of children *before* or *after* individualizing one more vertex. We may even distinguish cells *before* individualization in both cases if we include the decision of the cell selector into the invariant itself (i.e., using $(Q(G,\pi),\mathrm{Sel}(G,\pi))$ instead of $Q(G,\pi)$, which is only more expressive in case the cell selector is *not* quotient-graph-based).

In the following, we assume cell selectors are indeed quotient-graph-based. In the construction, we could indeed alternatively drop the additional restriction above with minor adjustments by allowing a more powerful cell selector.

For the remainder of this chapter, we say that a tree fulfills the *necessary conditions* if none of the conditions laid out by this section are violated.

3.5.2 Gadgets for Construction

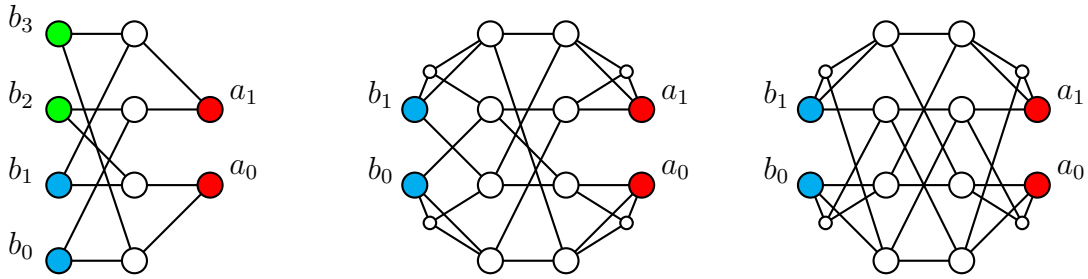
Given a colored tree (T,π) which satisfies the necessary conditions, our goal is now to construct a vertex-colored graph $G(T,\pi)$ and quotient-graph-based cell selector that yield the IR tree (T,π), up to renaming of colors. We make abundant use of gadget constructions, which we describe first.

All our gadgets have multiple *input* and *output gates*. Each gate is a pair of vertices that together form their own color class in the gadget. Vertices in the gates are the only vertices of the gadgets connected to other vertices outside the gadget. We say that vertices labeled with b_i denote the "input", while a_i denote the "output".

Gates can be *activated*, by which we mean the process of distinguishing the vertices of the gate pair into distinct color classes, and applying color refinement afterwards. We say activation *discretizes* the gadget if the resulting stable coloring on the gadget vertices is discrete.

We should note that three of the gadgets we are about to present (specifically the AND_i, unidirectional and dead end gadget) have already been used in other contexts related to color refinement [16, 10, 47].

AND_i **Gadget [16, 10, 47].** The AND_2 gadget, as illustrated in Figure 3.6a, realizes the logical conjunction of gates with respect to color refinement and an XOR gadget with respect to automorphisms.



(a) AND_2 gadget. (b) Unidirectional gadget. (c) Dead end gadget.

Figure 3.6: The AND_2 gadget and two variants of directional gadgets.

Given $i > 2$, we can realize an AND_i gadget with i input gates by combining multiple AND_2 gadgets in a tree-like fashion. The AND_i gadget is constructed by attaching the first and second input gate to an AND_2, whose output is connected to another AND_2 together with the third input gate, and so on. We use colors to order the input gates, i.e., we color the i-th input gate with color i.

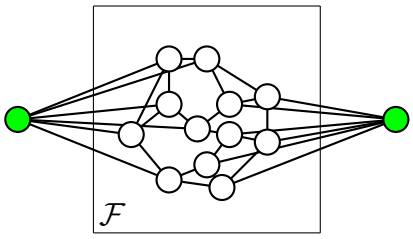
We define the special case of the AND_1 gadget to simply consist of a pair of vertices that functions as the input and output gate at the same time.

Lemma 45 ([47])**.** *The* AND_i *gadget admits automorphisms that flip the output gate and either one of the input gates while fixing other input gates. As long as some input gate remains unsplit, the output gate is not split, but activating all inputs discretizes the gadget.*

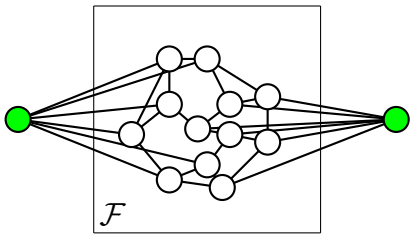
Unidirectional and Dead End Gadget [10, 47]. Next, we describe gadgets through which gate activation can be propagated or blocked depending on the direction of the gadget. Specifically, we construct the unidirectional gadget (Figure 3.6b) and the dead end gadget (Figure 3.6c). Note that the two gadgets are indistinguishable from each other by color refinement. The smaller vertices depicted in Figure 3.6 have been included to guarantee that the gadgets become discrete after the input and output gate have been split and can otherwise be ignored.

Lemma 46. *The unidirectional and dead end gadget are indistinguishable by color refinement. In the unidirectional case, activating the input discretizes the gate, but activating the output does not split the input gate. In the dead end case, both input and output have to be activated to discretize the gadget.*

Asymmetry Gadgets. Our next gadgets only have one gate (see Figure 3.7). Both of the asymmetry gadgets A_1 and A_2 (Figures 3.7a and 3.7b) have the crucial property that the two gate vertices of either gadget are initially indistinguishable by color refinement, but individualizing one of the gate vertices leads to a different quotient graph than individualizing the other gate vertex.

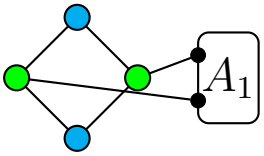


(a) The asymmetry gadget A_1.

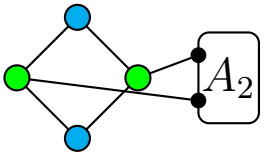


(b) The asymmetry gadget A_2.

Figure 3.7: Non-isomorphic asymmetry gadgets. The two input vertices are connected regularly to disjoint halves of the Frucht graph $\mathcal{F}$.



(a) A true edge.



(b) A fake edge.

Figure 3.8: The two types of concealed edge gadgets.

Lemma 47. *The asymmetry gadgets form asymmetric graphs that are stable under color refinement. Activating the input gate discretizes the gadget and we obtain two non-isomorphic colorings depending on which vertex was individualized. Furthermore,* $A_1 \ncong A_2$.

Concealed Edges. Lastly, we describe the *concealed edge gadget* that is used to hide edges from color refinement. The gadget has two vertices that represent the endpoints of an edge (the blue vertices in Figure 3.8). The idea is that instead of an edge connecting the two vertices, we insert a concealed edge gadget. For this the gadget has a pair consisting of two *inner vertices* (the green vertices in Figure 3.8), which are both connected to each input vertex. This pair is then connected to an asymmetry gadget. We define two classes of edges, where one type of edge attaches the asymmetry gadget A_1 and the other A_2. We call edges with asymmetry type A_1 *true edges*, and those with A_2 *fake edges*.

The crucial property is that as long as inner vertices of the gadgets are not distinguished, color refinement can not distinguish between true edges and fake edges. However, if we distinguish the inner vertices, true edges can indeed be distinguished from fake edges.

We always employ this gadget within the following design pattern. Whenever we want to connect two sets of vertices V_1 and V_2 with edges $E \subseteq V_1 \times V_2$ in a concealed manner, we first add a concealed edge gadget between *all* pairs $(v_1, v_2) \in V_1 \times V_2$. However, only if $(v_1, v_2) \in E$, we use a *true edge*, and whenever $(v_1, v_2) \notin E$ we use a *fake edge*. Finally, we connect all pairs of inner vertices of the concealed edge gadgets to some construction

that is used to *reveal* the edges.

The asymmetry gadget prohibits automorphisms from flipping the concealed edge gadget itself. However, care has to be taken when connecting the inner vertices to other constructions: it is imperative to connect the inner vertices of multiple concealed edge gadgets that are on the, say, left side of the asymmetry gadget, in the same manner. Otherwise, once revealed, edges could possibly be distinguished into even more categories than just fake and true edges.

3.5.3 Construction for Asymmetric Trees

For our construction, we first restrict ourselves to asymmetric trees, i.e., all leaves have different colors. Building on this, the following section takes symmetries into account. Let (T, π) be an *asymmetric*, colored tree that satisfies the necessary conditions (see Section 3.5.1).

We describe a graph $G(T, \pi)$ and a cell selector $S(T, \pi)$ such that (T, π) is (up to renaming of colors) the IR tree $T_{S(T,\pi)}(G(T, \pi))$. We describe the construction step by step. Initially, $G(T, \pi)$ is the empty graph and we successively add more and more vertices.

The goal is to model the graph and cell selector in such a way that there is a one-to-one correspondence between paths in T and sequences of individualizations in $G(T, \pi)$. Note that such sequences are precisely the paths in the IR tree $T_{S(T,\pi)}(G(T, \pi))$. To guarantee such a correspondence, certain properties of the paths in the tree T must translate into specific properties for their corresponding sequence of individualizations. When modeling $G(T, \pi)$, we must in particular ensure the following.

1. Two paths must end in nodes of different color exactly if the corresponding sequences of individualizations result in different quotient graphs.

2. A path must end in a leaf exactly if the corresponding sequence of individualizations (when followed by color refinement) results in a discrete coloring.

These two effects are guaranteed by different parts of our construction. We start by describing the part of the graph on which the cell selector operates, i.e., within which cells are chosen.

Selector Tree. One of the central difficulties is that color refinement executed on T may actually result in a coloring that is finer than π. This is precisely the reason why the tree must be concealed and why we cannot simply use the tree T itself. Therefore, structural and color information about T is encoded into the selector tree so that it is initially hidden from color refinement. In particular, the selector tree will be stable under color refinement, and only after individualizations are applied, parts of the structure of T are revealed.

To construct the selector tree, we first copy all the nodes of T and color each node with its level. To make cells appear uniform, we encode the edges of T in the selector tree using concealed edges, as follows. We fully connect nodes of level i to nodes of level

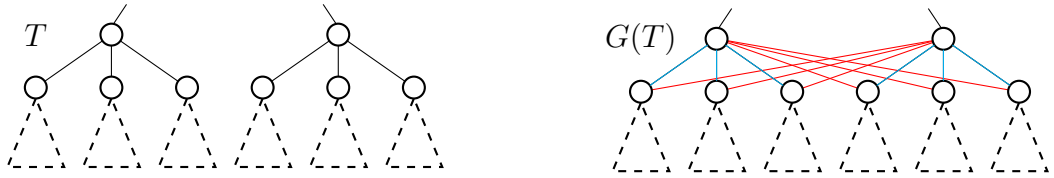


Figure 3.9: Connecting levels of the selector tree. Blue/red edges on the right symbolize true/fake edge gadgets

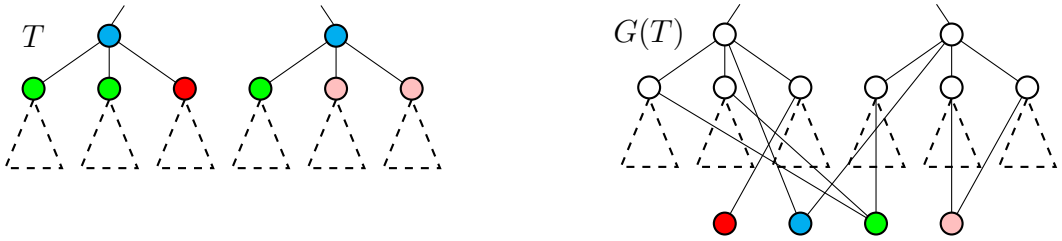


Figure 3.10: Colors of T are represented in $G(T)$ through concealed edges to special *color nodes*.

$i+1$ using concealed edges, creating a complete bipartite graph. Only if a node v at level $i+1$ is a child of node p at level i in T, we use a true edge between v and p. Otherwise, we use a fake edge. This guarantees that our copy of T is stable under color refinement. See Figure 3.9 for an illustration.

At some point, we will need to add another gadget construction to ensure that the edges between the levels are actually revealed at the right time. Assuming this for now, the cell selector $S(T, \pi)$ always chooses as the next cell the cell that consists of the children of the node chosen last. Here, children means children with respect to true edges in the selector tree.

Colors. Next, we translate the colors π of T into a construction that is part of $G(T, \pi)$. Recall that the colors indicate whether a sequence of individualizations should lead to differing quotient graphs. We make use of fake edges again to encode this: intuitively, we encode a one-to-one correspondence between selector tree nodes and their color in π using concealed edges. Since the edges are concealed, they are hidden from color refinement until revealed. We proceed level-wise. Let l be the level under consideration. Let C be the set of colors that appear at level l of (T, π). For all $c \in C$, we create a unique *color node* c in $G(T, \pi)$. This node is also colored with c. We now connect every node at level l of the selector tree to every node in C using concealed edges: we use a true edge for all pairs (n, c) where $\pi(n) = c$. All other edges are fake. See Figure 3.10 for an illustration.

As before, we still have to explain how and when edges are revealed. The idea is to always reveal the type of those concealed edges incident with node n at the point in time when node n is individualized.

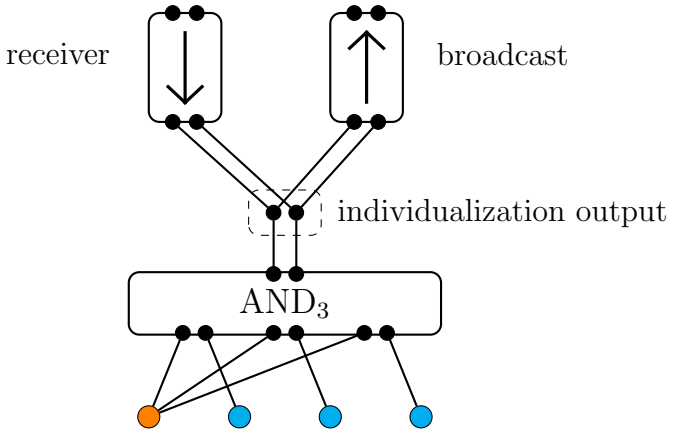


Figure 3.11: Leaf detection mechanism for the leftmost node of the cell. If the leftmost vertex is individualized, the AND_3 gadget is activated. The figure only shows true edges. In the overall construction, all remaining connections between vertices of cells at level l and AND_i gadgets of level l are fake edges.

Leaf Detection. Whenever we individualize a node that corresponds to a leaf in T, the graph $G(T, \pi)$ is supposed to become discrete, thereby terminating the IR process. The first step towards this is to add a construction that detects whether a specific node n in a cell was individualized. Then, a decision can be made as to whether n corresponds to a leaf or not. Let $s \geq 2$ be the size of the current cell (in the tree T the current cell is always the set of children of some node). For each vertex n in the cell, consider all $s-1$ (unordered) pairs with other vertices of the cell. We add an AND_{s-1} gadget and connect the left vertex of every input pair to n, and the other to one of the $s-1$ other vertices. An AND_{s-1} gadget is not symmetric in its input gates, so in order to keep things symmetrical, we actually add $(s-1)!$ many AND_{s-1} gadgets for every possible order of vertices in the input. We connect the output gates of all the AND_{s-1} gadgets to a new pair of vertices, which we call the *individualization output* of n.

Fact 48. *The individualization output is activated (i.e., split) whenever n is individualized. If the cell size is larger than 2, then the individualization output is not activated when another vertex in the cell is individualized.*

We should discuss the case of a size 2 cell, in which actually both vertices of the cell become singletons when one of them is individualized. The necessary conditions for T imply that either both vertices are leaves or both vertices are internal nodes in T (see Lemma 43). Hence, while this activates the construction for both vertices, the construction is still able to model any case that satisfies the necessary conditions.

We need to ensure the construction is stable under color refinement. Again, we can do so using concealed edges. Consider each level i in the selector tree: all of the aforementioned edges connecting vertices of level i in the selector tree with AND_{s-1} gadgets become true edges. We then insert fake edges between nodes of the selector tree of level i and the other AND_{s-1} gadgets of level i if there is no true edge. This way, the construction becomes

stable under color refinement.

Whenever a node does indeed correspond to a leaf, and its individualization output is activated, we want to propagate discretization to the entire graph. We add some control structures for every node n in the selector tree for this purpose. We add a unidirectional gadget if the node is a leaf in T, or a dead end gadget if not. We call this gadget the *broadcast* gadget of node n. We also add a *receiver* gadget to every node n, which is always a unidirectional gadget.

We connect the input of the broadcast gadget to the individualization output of n, as well as the output of the receiver gadget to the individualization output of n. Next, we connect the output of the broadcast gadget to the input of *all* receiver gadgets in the graph. See Figure 3.11 for an overview of the construction.

Fact 49. *When a leaf is individualized, in turn, all individualization outputs in $G(T, \pi)$ are split. As long as no leaf is individualized, individualization outputs are split only if they belong to individualized nodes.*

The idea goes as follows: if n is individualized, the individualization output is split. If n is a leaf, we want to propagate this split to all other individualization outputs, causing a discretization of the graph. For this, the broadcast gadget is activated, which sends the split to all the receiver gadgets, which in turn split their respective individualization output. If n is not a leaf, the broadcast gadget is a dead end gadget and activation of the individualization output does not have this effect. Below, we explain how we can use the same process to reveal cells of the entire selector tree as well as actual color nodes.

Revealing Cells and Colors. Recall that the cell selector makes choices along the selector tree and so choosing a particular cell corresponds to individualization of its parent node in the parent cell. Assume we are individualizing a node at level i of the selector tree. At this point, we want the connections in the selector tree from level i to level $i+1$ to be revealed. This is realized via the AND-gadget construction from the previous paragraph. We re-use the individualization output at level i to reveal the edges of the selector tree to level $i+1$. For this, we connect the output through a unidirectional gadget with the internal nodes of the concealed edges between level i and level $i+1$. To be precise, for every node n, we add a unidirectional gadget, the output of which is then connected to all internal nodes of the concealed edges. The use of unidirectional gadgets ensures that revealing the edges does not split an individualization output in the opposite direction.

Initially, the construction is stable under color refinement. Upon activating the unidirectional gadget, i.e., after a node on the previous level has been individualized, all true edges are distinguishable from fake edges. Hence, actual connections to cells are visible to color refinement.

Fact 50. *When a node at level i is individualized its color and its edges to level $i+1$ are revealed. Before individualizing a node at level $i+1$, these are the only revealed edges connected to level $i+1$.*

In order to actually activate the individualization output, we also need to reveal edges from level $i+1$ nodes to the AND_{s-1} gadgets. Hence, we do the same construction as

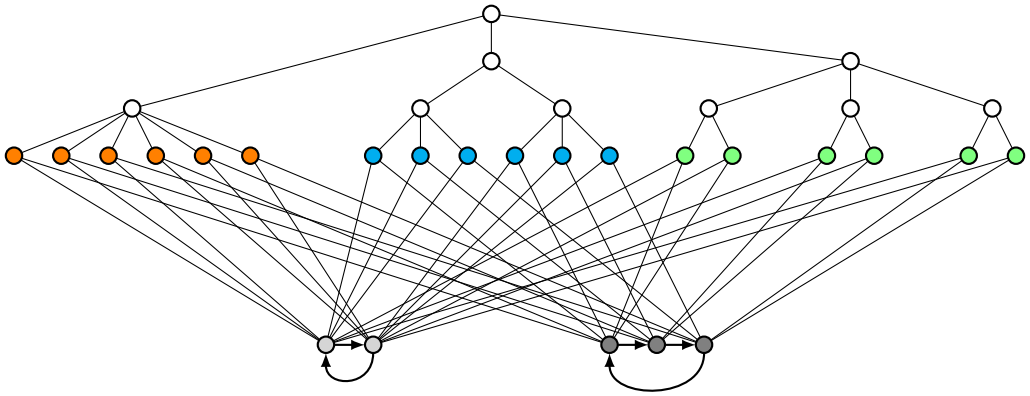


Figure 3.12: Symmetry cycles couple leaf orbits across multiple branches of the selector tree. The illustration omits fake edges. In the construction, cycles do not contain directed edges, but specially colored nodes that indicate direction.

above, connecting the unidirectional gadgets we added on level i to reveal these edges on level $i + 1$.

Note that the construction guarantees that if two nodes n_1, n_2 at level l of T have a different number of children, then n_1 and n_2 are distinguished. This reflects the necessary requirement discussed in Lemma 44. As mentioned there, this restriction could be avoided through the use of a more powerful cell selector.

For the very first level of the selector tree, the immediate children of the root, we remove the concealed edge construction by removing fake edges, such that the level is initially revealed.

Finally, the same technique is also used to reveal colors. We connect the individualization output of node n at level l to the inner vertices of the concealed edges between n and the color nodes C of level l. This immediately reveals the color of n whenever we individualize n. In this case, we need no special construction for level 1.

3.5.4 Construction with Symmetries

We expand our construction so that it can also handle colored trees (T, π) with prescribed symmetries. As such, the graph $G(T, \pi)$ can also be built from a tree (T, π) that is not necessarily asymmetric. In this case, sequences of individualizations along root-to-leaf paths still produce the desired tree (T, π) as a subtree of $T_{S(T,\pi)}(G(T, \pi))$. However, $G(T, \pi)$ is supposed to become discrete after the IR process reaches a leaf of (T, π), but at this point the selector tree in $G(T, \pi)$ is only split up to orbits that correspond to orbits of T.

Discretization of orbits is challenging since we need to make sure that the symmetries are not destroyed by the addition of new gadgets. Once leaf orbits have been discretized, discretization propagates through the selector tree as before and the whole construction becomes discrete.

Overall we need to construct the graph H_T mentioned in Theorem 38.

To construct H_T, we introduce *symmetry cycles* and *symmetry couplings*. The basic idea is shown in Figure 3.12, a detailed explanation follows below. This in turn defines a new construction $\tilde{G}(T, \pi)$ by adding a concealed version of H_T to the selector tree.

Discretization up to Orbits. Revealing the true and fake edges in $G(T, \pi)$ is not enough to discretize orbits, since this just reveals the orbit partition. By definition, nodes in the same orbit must be connected to the rest of the construction in a symmetric way and thus, splitting an orbit has to be induced by individualizations inside the orbit or through connections to other orbits that have already been split.

We thus face two independent problems related to leaf orbits. First, when the IR process on $G(T, \pi)$ reaches leaf l, the orbit of l may not be discrete in the current construction. Second, other leaf orbits have not been split at all. We solve these problems in an isolated setting first, by providing a constructive proof of Theorem 38. We then add the graph H_T of the construction on top of $G(T, \pi)$ to obtain our final construction $\tilde{G}(T, \pi)$.

For now, we are in the setting of Theorem 38. We first describe how to construct H_T from T.

Symmetry Cycles. Consider a leaf orbit Ω in T. Let $p_1 p_2 \dots p_m = |\Omega|$ denote a prime factorization. We construct directed cycles of length p_i for $i \in \{1, \dots, m\}$, such that we have one cycle for each prime p_i. Cycles of the same length are ordered, which is expressed by giving them distinct colors.

To model a directed edge, we employ two colored vertices. We add two special vertex colors d_1, d_2 for this purpose. A symmetry cycle of size n consists of n base nodes and $2n$ edge nodes, of which n are colored with d_1 while the other n are colored with d_2. We define an arbitrary order on the base nodes $b_1, \dots, b_n$, d_1-colored edge nodes $d_{1,1}, \dots, d_{1,n}$ and d_2-colored edge nodes $d_{2,1}, \dots, d_{2,n}$. The cycle is then connected up by attaching b_i to $d_{1,i}$, $d_{1,i}$ to $d_{2,i}$ and $d_{2,i}$ to b_{i+1} for all $i \in \{1, \dots, n\}$ (we set $b_{n+1} = b_1$).

Symmetry Coupling. The next step of the construction is to match leaf orbits with symmetry cycles (see Figure 3.12). This naturally restricts the possible symmetries of leaf orbits but we can choose the connections in a consistent way that does not break up any orbits.

The pairwise matching of leaf orbits is realized by coupling each orbit with the set of symmetry cycles. Thus, it is enough to describe a coupling between one leaf orbit Ω and the set of symmetry cycles. To this end, we first introduce a new tree T_Ω.

Consider the common ancestor a of Ω in T that has least distance to Ω. The root of T_Ω is a and T_Ω contains exactly those a-to-leaf branches of T that end in Ω. Then T_Ω describes the group structure of the symmetries of Ω that correspond to automorphisms of T. Note that root-to-leaf branches of T_Ω can be permuted transitively. In particular, the degree of T_Ω is uniform for each level. Then sibling classes on the same level have the same size and this size always divides $|\Omega|$. Note that the sibling class size may actually be 1 for some levels.

We modify T_Ω into another tree T'_Ω whose sibling class sizes are prime numbers. The first modification is to iteratively contract levels of T_Ω if the branching factor between them is 1. This removes sibling classes of size 1. Next, consider the i-th level of T_Ω and assume the sibling class size on level i is a compound number, say $s = rp$ for a prime p and $r > 1$. We add a new level between levels i and $i-1$ by partitioning each sibling class on level i arbitrarily into p classes of size r. We repeat the process exhaustively to obtain T'_Ω.

Let $|\Omega| = p_1 \cdots p_m$ be a prime factorization, then the multiset of branching factors in T'_Ω is given by $\{\!\{p_1, \ldots, p_m\}\!\}$. Furthermore, each permutation of leaves corresponding to an automorphism of T'_Ω also defines an automorphism of T, since both types of modifications we described only restrict the possible symmetries but they do not break up orbits: contracting levels with branching factor 1 does not interfere with automorphisms at all and when partitioning sibling classes into equally sized blocks, the action on each sibling class remains transitive. Therefore, the leaves of T'_Ω still form one orbit.

We use T'_Ω to define a coupling between leaves in T and symmetry cycles.

For each sibling class C on level i of T'_Ω, we connect the descendants of C to a symmetry cycle (whose length is the sibling class size of level i), such that leaves are connected to the same vertex of the cycle if and only if they descend from the same node in C. In particular, sibling classes of leaves are connected to symmetry cycles via a perfect matching. We always use one fixed symmetry cycle for each level. Recall that symmetry cycles of the same length are ordered. For all orbits, we always use the first cycle of length p for the highest level with sibling class size p and so on. This ensures that we do not introduce dependencies on rotations of different symmetry cycles (different orbits in T might have ancestors in a common orbit).

Proof of Theorem 38. We construct H_T from T by attaching T'_Ω to each leaf orbit Ω in T, such that we identify leaves of T'_Ω with nodes in Ω. We choose a color that is not contained in T to color inner vertices of T'_Ω. Then we add symmetry cycles to H_T (as a disjoint union) and connect the symmetry cycles with each T'_Ω as described in the construction above. Again, we use new colors for each symmetry cycle. Thereby we make sure that $\mathrm{Aut}(H_T)$ fixes the copy of T as well as each symmetry cycle and each T'_Ω setwise.

By construction, $\mathrm{Aut}(T'_\Omega)$ acts on Ω as a transitive subgroup of $\mathrm{Aut}(T)|_\Omega$ (automorphisms restricted to Ω). Consider the graph $H'(\Omega)$ induced by H_T on T'_Ω and the set of symmetry cycles. Let level i of T'_Ω be connected to a symmetry cycle C_{p_i}. Observe that a rotation of C_{p_i} induces a simultaneous cyclic permutation in all sibling classes on level i and that in $H'(\Omega)$ different symmetry cycles can be rotated independently from each other. Moreover, all automorphisms of H' are induced by rotations of symmetry cycles and since all sibling classes of T'_Ω can be permuted transitively, $\mathrm{Aut}(H'(\Omega)) \leq \mathrm{Aut}(T)_{|\Omega}$ acts regularly on Ω.

Since we choose the order of symmetry cycles of the same length consistently for all orbits, we do not introduce dependencies between symmetry cycles, even in the full construction H_T. This finally implies that the action of $\mathrm{Aut}(H_T)$ on Ω is permutation isomorphic to the action of $\mathrm{Aut}(H'(\Omega))$ on Ω, in particular, the action on the full set of leaves is semiregular. □

Discretization of Orbits. To build $\tilde{G}(T, \pi)$, we now add the construction from Theorem 38 to the selector tree in $G(T, \pi)$. Observe that each leaf of T'_Ω is connected to exactly one vertex in each symmetry cycle. That means that individualization of a leaf in $\tilde{G}(T, \pi)$ individualizes a node in each symmetry cycle and in turn, all symmetry cycles become discrete. Moreover, since leaves that are not siblings have predecessors that are siblings in some higher level, for each pair of leaves there is one symmetry cycle such that the leaves are connected to different nodes of the cycle. As a consequence, individualizing a leaf in $\tilde{G}(T, \pi)$ discretizes all symmetry cycles which then distinguishes all leaves from each other.

Fact 51. *Individualization of a root-to-leaf path in $\tilde{G}(T, \pi)$ discretizes the set of leaves.*

Concealing Symmetry Couplings. We need to hide H_T from color refinement until a leaf is individualized, or otherwise, leaves would be distinguishable from internal nodes in the selector tree. As before, we do so by employing concealed edges. In the construction of H_T, we replace all edges with true edge gadgets. Then, to conceal the edges, all pairs (n, v), where v is contained in a symmetry cycle and n is a node in the selector tree, which are not yet connected by a true edge gadget, are connected with a fake edge. The type of these edge gadgets is revealed upon activating a (unidirectional) broadcast gadget. For this we connect the inner nodes of the concealed edge gadgets to the output of all broadcast gadgets.

3.5.5 Necessary Conditions are Sufficient

In this section, we prove the correctness of our graph constructions $G(T, \pi)$ and $\tilde{G}(T, \pi)$. We start by proving some more specific properties, which ultimately culminate in our main theorem.

Throughout the section, if v is a node of the selector tree in $G(T, \pi)$, then v_T denotes its corresponding node in T.

Lemma 52. *The selector tree in $\tilde{G}(T, \pi)$ is stable under color refinement.*

Proof. Initially, vertices in the selector tree are colored with their level. Recall that by our concealing paradigm, all connections of the selector tree are hidden from color refinement and nodes on the same level are connected to the same combined number of true or fake edges. This immediately implies the claim. □

Lemma 53. *Let (T, π) be asymmetric and let l be a leaf in the selector tree. If the concealed edges connecting l to its corresponding* AND*-gadgets have been revealed, $G(T, \pi)$ becomes discrete after individualizing l and applying color refinement.*

Proof. Since we assume concealed edges to the respective AND-gadgets have been revealed, individualizing l, by construction, splits the vertices of its corresponding individualization output. Since l is a leaf in the selector tree, the connected broadcast gadget is a unidirectional gadget. The gadget is connected to all inputs of receiver gadgets in the

graph. Hence, the split is propagated and all individualization outputs in the graph are split.

Now, the individualization outputs, in turn, reveal all edges in the selector tree, as well as concealed edges to color nodes. Since T is asymmetric, the connections to the color nodes, in turn, discretize nodes in the selector tree that correspond to leaves of T.

Since we also reveal all edges of the selector tree itself, all nodes in the selector tree subsequently become discrete. This fully discretizes the attached AND_i gadgets as well as their connected individualization outputs. Note that at this point, for any broadcast gadget, even if they are a dead end gadget, all inputs and outputs are discrete, meaning the gadgets themselves become discrete as well.

Since all nodes belonging to sets connected by concealed edges are now discrete, and all edges have been revealed, the concealed edge gadgets now become fully discrete as well.

This, in turn, covers all of the constructions in $G(T, \pi)$. □

Lemma 54. *Consecutive choices of the cell selector on $G(T, \pi)$ correspond to sibling classes along paths of T.*

Proof. Initially, v is colored with the level of v_T. In particular, nodes corresponding to the first level of T form a color class in $G(T, \pi)$ that is stable under color refinement (see Lemma 52). Hence, it is, by definition, the first class the cell selector chooses.

In case v_T is a leaf of the tree, the graph becomes discrete. This implies there is no subsequently selected cell. Hence, we can assume v_T is an inner node of the tree.

By definition, whenever a node v is individualized, the next cell chosen by the cell selector corresponds to children of v_T. Recall that v is connected to other nodes of the selector tree via true edge gadgets if and only if they correspond to children of v_T and v is connected to all other nodes of the selector tree via fake edge gadgets. By construction, individualizing v activates the individualization output of v.

Since v is an inner node, this split does not propagate into other gadgets: the receiver gadget is a unidirectional gadget in the wrong direction. This gadget therefore does not propagate the split. Furthermore, the broadcast gadget is a dead end gadget. Note that the AND, receiver, and broadcast gadgets attached to v can be distinguished from the other gadgets of their respective type. However, none of these splits propagates further since all the other gadgets are connected uniformly to v and the gadgets of v.

The individualization output does, however, reveal the edges in the selector tree that connect v to its children: after individualizing v, fake edge gadgets attached to v are distinguished from true edge gadgets attached to v and so the next cell can be chosen among children of v.

It remains to argue that at this point, children of v are indistinguishable in $G(T, \pi)$. We may inductively assume that edge types between higher levels have not been revealed yet. Thus, since edge types are initially indistinguishable by color refinement, the only relevant connections children of v have, are connections to the layer of v and to inputs of AND_i gadgets. Both of these connections are uniform by construction. □

Lemma 55. *Consider two nodes v, w in the selector tree of $G(T, \pi)$. If $\pi(v_T) \neq \pi(w_T)$ then individualization of v and w, respectively, produces different quotient graphs.*

Proof. Individualizing v or w also activates their corresponding individualization output, which in turn reveals the concealed edges that connect v or w to the color nodes. In particular, the corresponding quotient graphs already differ with respect to these connections since $\pi(v) \neq \pi(w)$. □

Lemma 56. *Consider nodes v, w in the selector tree of $G(T, \pi)$ such that v_T and w_T are leaves of T. If $\pi(v_T) = \pi(w_t)$ then v and w can be mapped to each other via automorphisms of $G(T, \pi)$. The same holds for $\tilde{G}(T, \pi)$.*

Proof. Recall that Lemma 40 implies that the equally colored leaves v_T and w_T lie in the same orbit of T. The selector tree without connections to individualization outputs or symmetry coupling is just a concealed copy of T, where edges and non-edges were replaced by true edge gadgets and fake edge gadgets, respectively and colors were replaced by true/fake connections to color nodes. Thus, automorphisms of (T, π) are in one-to-one correspondence with automorphisms of the subgraph induced on the isolated selector tree together with color nodes. By construction, two nodes in a common cell are connected uniformly to individualization outputs belonging to their cell or their common parent cell. Thus, all automorphisms of the selector tree induce automorphisms of $G(T, \pi)$ by permuting individualization outputs (and the corresponding gadgets) accordingly. Finally, from Theorem 38, we obtain that the leaf orbits of $G(T, \pi)$ are the same as the leaf orbits of $\tilde{G}(T, \pi)$. □

Lemma 57. *Consider two nodes v, w in the selector tree of $G(T, \pi)$. If $\pi(v_T) = \pi(w_T)$ then individualizing nodes along paths to v and w, respectively, produces the same sequence of quotient graphs.*

Proof. First, recall that due to Lemma 42, color classes in T are contained within single layers. This implies that v and w belong to the same level l of the selector tree and, in particular, they are connected to the inputs of AND_i gadgets uniformly. We make a case distinction on whether v_T and w_T are leaves or not.

Assume v_T and w_T are inner nodes of T. Individualizing v or w reveals the concealed edges connecting them to color nodes. However, by assumption, they are connected to the same color node.

Furthermore, individualizing v or w reveals the concealed edge gadgets connecting level l to $l + 1$ in the selector tree. By Lemma 42, v and w have the same number of children on level $l + 1$. Furthermore, the concealed edges connecting the children to other parts of the graph are not revealed. In particular, their color has not been revealed. Hence, they are still indistinguishable.

Also, due to Lemma 42, nodes of the same color in T have predecessor nodes that are of the same color level-wise and have the same number of children. In case v_T is not a leaf, the latter implies that v and w are uniformly connected in all steps of the construction. Since edges have only been revealed up to the level of v and w, this shows the equality of quotient graphs.

If v_T and w_T are leaves, we can apply Lemma 56. Note that actually the complete root-to-leaf paths for v_T and w_T are in the same orbit and, thus, individualizations along

both paths produce isomorphic quotient graphs by the isomorphism invariance of color refinement. □

Lemma 58. *Let v correspond to a node in the selector tree that belongs to a leaf of T. If the concealed edges connecting v to its corresponding* AND*-gadgets have been revealed, $\tilde{G}(T, \pi)$ becomes discrete after individualizing a root-to-v path and applying color refinement.*

Proof. Consider a node l in the selector tree for which l_T is a leaf of T. Recall that $\tilde{G}(T, \pi)$ is just $G(T, \pi)$ extended by symmetry cycles and symmetry coupling. In particular, as in the asymmetric case, individualizing l will reveal the colors of nodes in the selector tree (see the proof of Lemma 53). In particular, since leaf colors correspond to orbits, color refinement partitions the leaves into their orbits. Furthermore, all edge types are revealed at this point and, thus, from a combinatorial perspective, we may treat true edge gadgets as edges and fake edge gadgets as non-edges.

By Fact 51, individualization of a path to v induces the complete discretization of the set of leaf nodes in the selector tree and, as in the asymmetric case, this discretizes the whole construction. □

We are now ready to prove our main theorem:

Theorem 59. *Let (T, π) be a colored tree that fulfills the necessary conditions. Then, $T_{S(T)}(\tilde{G}(T))$ is equal to (T, π) (up to renaming colors).*

Proof. By Lemma 52, the selector tree in $\tilde{G}(T, \pi)$ is initially stable under color refinement and, in particular, its levels form stable color classes. The cell selector chooses the first level of the selector tree as the first cell to individualize. By Lemma 54, the subsequent choices are always given by the full set of children of the node last individualized. Together with Lemma 58, this implies that the tree structure of the IR tree $\Gamma := T_{S(T)}(\tilde{G}(T))$ is exactly the same as the structure of T and we obtain a one-to-one correspondence between Γ and T.

Finally, the Lemmas 55 and 57 together show that nodes in Γ obtain the same color (i.e., the sequences of individualizations they describe give the same quotient graphs) if and only if the corresponding nodes in T have the same colors, so up to renaming colors Γ and T are the same tree. □

Chapter 4

Color Refinement

Color refinement is repeatedly and continuously applied in IR algorithms. Its efficiency is crucial when designing a fast practical graph isomorphism solver. Modern implementations are based on Hopcroft's algorithm for automata minimization [55], which was first adapted to color refinement by McKay in NAUTY [74]. Given a graph, color refinement iteratively recolors the vertices producing increasingly fine partitions of vertices into color classes. Starting with an initial, usually monochromatic coloring, in each iteration the colors of the vertices are chosen to depend on the colors of the neighbors and their multiplicities. If vertices differ in the number of neighbors they have in some color class, the algorithm *splits* up the vertices accordingly by assigning them distinct colors. This is done exhaustively until no further splits are possible.

Indeed, the best known upper bound for color refinement is $\mathcal{O}((n+m)\log(n))$ (see [16]). Remarkably, within a model with modest assumptions, a tight lower bound construction matching this upper bound was given in 2015 [16]. This result tells us that there are graphs for which color refinement, if it is implemented within these modest assumptions, runs in time $\Omega((n+m)\log(n))$. However, the result does not make any comparative statements between various ways to implement color refinement. In fact, there are dramatic differences in the various implementations of color refinement. While all color refinement algorithms depend on performing the aforementioned splits there is a lot of freedom as to which order we perform the splits in, or how the splits themselves are exactly implemented. Moreover, when considering its context within the individualization-refinement framework, implementations usually perform various additional tasks during color refinement itself.

Efficient Color Refinement. We begin with an extensive exposition of an efficient color refinement implementation. The goal of this exposition is to provide a description of the algorithm as implemented in DEJAVU. The algorithm is reverse-engineered from the implementation of TRACES. We argue its correctness and worst-case running time of $\mathcal{O}((n+m)\log(n))$. Furthermore, we discuss differences between our version, the implementation of TRACES, as well as the theoretical algorithm used in the upper bound given in [16].

Worklist Choice. It turns out that color refinement comes with many potential design choices. A central choice is in which *order* splits of color refinement should be performed. Potential splits are usually kept inside a *worklist*. The order is therefore usually determined by the choice of the worklist data structure. Intuitive choices include a stack, queue, priority queue or combinations of these.

So far however, there has been no rigorous analysis whether one worklist choice is superior over another – or how significant the order of splits actually is. Going one step further, a natural question is whether there are efficient optimal solutions. If not the case, maybe there are at least solutions that are competitive with all other methods. In particular, we provide a rigorous formal analysis for worklists used in color refinement.

We employ a two-pronged approach. We distinguish (1) algorithms that may only use information realistically collected during the color refinement process itself, and (2) algorithms that are allowed to compute additional information about the underlying graph. Remarkably, our results in the two orthogonal models concur in their conclusion. Namely, that there is no design choice that is competitive beyond a logarithmic factor.

More specifically, in (1) we model algorithms that may only access information explored during the color refinement process itself. For this we define a formal online model within which, in fact, all practical algorithms operate. In this model, the algorithmic decisions of when to refine with respect to what may solely depend on this information. We prove that this information does not suffice to make optimal or even competitive choices, no matter the amount of computational power used. Specifically, we show no online algorithm is within a logarithmic factor of the offline optimum.

For (2), we define an "offline" version of the problem, which is essentially to compute an optimal split order for a given graph. Through a reduction from the set cover problem we prove an approximation hardness result. Specifically, unless $\mathsf{P} = \mathsf{NP}$, no approximation factor in $o(\log(n))$ can be achieved by polynomial time algorithms. This proves that unless $\mathsf{P} = \mathsf{NP}$, even when collecting more information about the underlying graph than current algorithms actually do, computing a competitive let alone optimal order of splits is intractable.

Splits. Another important practical design choice is how to actually perform the *splits* of color classes. A split routine counts the neighbors of a given color class, and in turn splits up the neighboring color classes according to these counts. In practice, the split routine usually contains the *hot spot* of the entire IR algorithm. Therefore, the split routine is a natural target for low-level optimizations.

Algorithms have long featured a simplified, faster split routine for *singleton* color classes. However, Traces pioneered the approach of further varying the splits depending on the density of the graph and color class. We give a detailed account of the different split routines used in our implementation, and reason why the different split routines yield the appropriate theoretical runtime.

IR Context. Lastly, we describe further optimizations and techniques used in an IR context which are related to color refinement. We also describe how they can be efficiently used within the algorithm described in this section. This includes *undoing* a color refinement, the *trace invariant* as introduced by Traces, *early-out* opportunities of the algorithm, as well as the so-called *matched vertex colorings* as introduced by saucy.

4.1 Efficient Color Refinement

We begin with the description of an efficient color refinement routine, as is provided in Algorithm 9, Algorithm 10, Algorithm 11, and Algorithm 12.

Intuition behind Algorithm 9. The basic idea is as follows. If two vertices in some class X have a different number of neighbors in some class C then X can be split by partitioning it according to neighbor counts in C. Whenever we split up a class X according to its connections to another class C in such a fashion we say that we *refine* X *with respect to* C, which is the subroutine described in Algorithm 10. Specifically, this means that after the split, two vertices have the same color precisely if they had the same color before the split and they have the same number of neighbors in X. We repeatedly split classes with respect to other classes until no further splits are possible, or in other words until the worklist W of Algorithm 9 is exhausted. As defined in Chapter 2, a partition not admitting further splits is called equitable.

Note that in Line 27 of Algorithm 10 the method makes a call to a "`ReportSplit`" routine. This routine is *not* part of the color refinement algorithm itself, but will be used in the following sections to implement further routines based on color refinement. For now, you may assume that this routine has no effect.

Description of Algorithm 9. The first few steps of the algorithm initialize data structures that are used throughout the algorithm and its various subroutines. We describe them once they are needed.

(Initialization.) The first important step is the initialization of the *worklist* W. The worklist contains colors of π which still have to be considered by the algorithm. Initially, the worklist W is filled with all the colors of π, in increasing order. Note that in the arguments below we do not specify exactly how the worklist operates. Indeed, in Section 4.2, we thoroughly analyze how the choice of a specific worklist may affect the running time of the algorithm. Intuitively, for now, the reader may simply think of the worklist as a stack.

(Main Loop.) Next, we enter the main loop of the algorithm: we pop a color c from the worklist W, and run Algorithm 10 on c. The main loop is repeated until W is empty. Note that Algorithm 10 may enqueue more elements to W. We therefore now continue with the description of Algorithm 10.

(Neighbor Counts.) The goal of Algorithm 10 is to look at all the neighbors of vertices of $C = \pi^{-1}(c)$, and split their color classes according to the neighbor counts in c. Hence, the algorithm begins by iterating over the vertices of C (Line 3). For each of the vertices $v \in C$, it then iterates over all of the neighbors $N(v)$ (Line 4). In the following we may also refer to this overall process as iterating over the neighbors of C.

For each of the neighbors v' of c, we then do the following. First, we check whether we have seen the color x of v' before. If not, we record the color into Col_{old}.

Next, we check whether we have seen v' itself before. If not, we record the vertex into a list of neighbors in x (Line 9). Intutively, the arrays $\mathrm{Adj}_{\mathrm{Col}}$ and $\mathrm{Num}_{\mathrm{Col}}$ together

maintain efficiently accessible lists of adjacent vertices for each color c', where adjacent means adjacent to some vertex of C.

Lastly, we increment the degree of v' in $\mathrm{Deg}[v']$. Having finished the loop beginning in Line 3, we record the following facts:

Fact 60. *After the loop beginning in Line 3 is finished in Algorithm 10, the following is true:*

1. *For each $v \in V(G)$, $\mathrm{Deg}[v]$ contains the number of neighbors of v with color c.*
2. *The list Col_{old} contains precisely those colors x, which contain a vertex v' that is adjacent to some vertex v with color c.*
3. *For each color $x \in \pi(V(G))$, $\mathrm{Num}_{\mathrm{Col}}[x]$ contains the number of vertices with color x that are adjacent to a vertex with color c.*
4. *For each color $x \in \pi(V(G))$, when $\mathrm{Num}_{\mathrm{Col}}[x] \geq 1$ holds, the elements denoted by $\mathrm{Adj}_{\mathrm{Col}}[x], \ldots, \mathrm{Adj}_{\mathrm{Col}}[x + \mathrm{Num}_{\mathrm{Col}}[x] - 1]$ are vertices with color x that are adjacent to a vertex with color c. When $\mathrm{Num}_{\mathrm{Col}}[x] = 0$ holds, there are no such vertices.*

(Sort.) In the next step, we remove colors from Col_{old} that will not split according to the counts in Deg. Then, we sort the remaining elements of Col_{old}.

(Spacious Colors.) For each color $x \in \mathrm{Col}_{old}$ that *will* split, we then perform the following operations.

The algorithm makes a list of the different degrees $\mathrm{Deg}[v]$ that occur for vertices $v \in \pi^{-1}(x)$. This is recorded in the list UniqueDeg. Note that as the name suggests, UniqueDeg only contains unique degree values. These values correspond to the color classes that we will split x into. Additionally, for each different degree d in UniqueDeg, $\mathrm{Deg}_{\mathrm{Col}}$ maintains how many vertices v of x have $\mathrm{Deg}[v] = d$. This corresponds to the *size* of color classes that x will be split into.

Next, for each degree $d \in$ UniqueDeg, we compute precisely what the new color will be. Note that this makes use of the fact that π is a spacious coloring (see Section 2.2.4). If there are vertices of x, that are not connected to c, these will remain in the color x. So for the first new color, we leave room in the coloring for these disconnected vertices (Line 23). Then, we iterate over the degrees in increasing order, and accumulate the number of vertices of each degree. In each iteration, we determine one new color (Line 26). To be more precise, we create a mapping from the degrees to the new color.

Now that we have this mapping, we are ready to rearrange the coloring such that the split is actually performed. This is described in Algorithm 11. The details are however of no particular importance, the algorithm is mainly concerned with maintaining the properties of the stored coloring.

More crucially, once the split is performed, we want to maintain our worklist, which is described by Algorithm 12. The idea is that we enqueue all but the (first) largest color x_i. And if the current color x is already on the worklist W, we can replace it with x_i. We argue why this is sufficient further below.

This concludes our description of the color refinement algorithm.

Correctness of Algorithm 9. We want to argue three properties:

1. The result π' of the algorithm is an equitable coloring of (G, π).
2. The color partition of the result π' is coarser than the orbit partition.
3. The result is isomorphism invariant, i.e., for all $\varphi \in \mathrm{Sym}(V(G))$, given (G^φ, π^φ), the algorithm computes π'^φ.

Claim (1). Assume towards a contradiction there are v and v' with $\pi'(v) = \pi'(v')$, but there is a non-trivial color $c \in \pi'(V(G))$ such that

$$N(v) \cap \pi'^{-1}(c) \neq N(v') \cap \pi'^{-1}(c).$$

Let c be the color with this property which appeared earliest in the execution of Algorithm 9. Since $C = \pi^{-1}(c)$ could not have been added to the worklist, C was the largest fragment of a color class D, split into $C_1, \ldots, C_k$ (with respective colors $c_1, \ldots, c_k$), where $C = C_j$ for some $j \in \{1, \ldots, k\}$. Since C was the earliest color appearing in the algorithm with the above property,

$$N(v) \cap D = N(v') \cap D$$

follows. We observe that

$$\sum_{i \in \{1,\ldots,k\}} N(v) \cap C_i = N(v) \cap D = N(v') \cap D = \sum_{i \in \{1,\ldots,k\}} N(v') \cap C_i,$$

and

$$\sum_{i \in \{1,\ldots,k\} \setminus \{j\}} N(v) \cap C_i \neq \sum_{i \in \{1,\ldots,k\} \setminus \{j\}} N(v') \cap C_i.$$

hold. Therefore, there must be a $C_i \neq C$ with

$$N(v) \cap C_i \neq N(v') \cap C_i,$$

where the corresponding color c_i must have been added to the worklist. Hence, v and v' must have been split with respect to c_i, a contradiction to the assumption that $\pi'(v) = \pi'(v')$.

Claim (2). Vertices v, v' only ever obtain different colors, if there is a color class C such that

$$N(v) \cap C \neq N(v') \cap C.$$

Hence, an automorphism can not preserve the neighborhoods when mapping v to v', meaning v and v' can not be in the same orbit of the initial graph.

Claim (3). Clearly, determining splits of the classes of vertices according to other classes of vertices (when interpreted as sets), is an isomorphism-invariant process. Sorting of the degree sequence Line 22 and previous colors Line 13 ensures that these classes always obtain the same exact color, which in turn also ensures an isomorphism-invariant order of the worklist, and in turn the entire algorithm.

Runtime of Algorithm 9. We argue that Algorithm 9 can be implemented such that in runs in worst-case time $\mathcal{O}((n+m)\log n)$.

Let us first record the runtime of the subroutines: first of all, assuming all operations of the worklist run in time $\mathcal{O}(1)$, Algorithm 12 can be implemented in time $\mathcal{O}(k)$, where k refers to the list of colors in the input. We require that the data structure of the worklist enables the efficient look-up and replacement of colors already in the worklist. This can usually be achieved by storing an array that provides pointers from colors to a position in the worklist. Next, Algorithm 11 only consists of three loops where each loop runs in time $\mathcal{O}(l)$, where l refers to the list of vertices in the input.

The runtime of Algorithm 10 is more interesting. Let us first make a runtime analysis of the algorithm ignoring the two sorting operations (Line 13 and Line 22). We will come back to these operations further below.

Ignoring the sorting operations, it is easy to see that the two loops of Line 3 and Line 4 run in time that is linear in the number of vertices of $\pi^{-1}(c)$ and edges incident to the vertices $\pi^{-1}(c)$, i.e.,

$$\mathcal{O}\left(\sum_{v\in\pi^{-1}(c)} 1+|N(v)|\right).$$

Indeed, the rest of the algorithm runs at most in linear time in the lists recorded during this operation, so this is indeed the overall time of the routine.

Let us now argue the runtime of the overall procedure, i.e., Algorithm 9. For this, we need to analyze the behavior of the worklist W. First, let us observe that a given vertex v can only ever be re-introduced to the worklist, if the color c that contains v is split. However, if a color class $C = \pi^{-1}(c)$ is split, then only all but the largest fragment is enqueued to the worklist (see Algorithm 12). This means, all the fragments $C_1, \ldots, C_k$ of C that are *actually enqueued* are at most half the size of C, i.e., $|C_i| \leq \frac{|C|}{2}$ for all $i \in \{1, \ldots, k\}$. From the perspective of a vertex v, this means each time it is re-introduced to the worklist, the color that contains it can at most be half the size of the previous color. Hence, it can at most be re-introduced to the worklist $\log n$ number of times. This means, each vertex can at most be considered $\log n$ number of times in a color c of Algorithm 10, which gives us a total running time of

$$\mathcal{O}\left(\left(\sum_{v\in V(G)} 1+|N(v)|\right)\log n\right) = \mathcal{O}((n+m)\log n),$$

barring the sorting operations.

Lastly, we now argue that the two sorting operations (Line 13 and Line 22) are within the appropriate running time. Let us recall a crucial argument of [16]. Overall, the algorithm is only ever able to produce n *new* colors: since we are relying on spacious colorings, there can only ever be a total of n different colors.

In each iteration i of the main loop, a total of k_i new colors may be introduced, which together sum up to at most n. Hence, assuming there are l total iterations we record the

Algorithm 9: Detailed version of the color refinement algorithm.

```
1  function ColorRefinement
       Input: ➢ graph G = (V, E)
              ➢ coloring π
      Output: ≺ refines coloring π
2      // initialize auxiliary workspace for subroutines
3      initialize reset array Deg and Num_Col with length n;
4      initialize small integer sets Col_old and UniqueDeg with length n;
5      initialize arrays Adj_Col and Deg_Col with length n;
6      // initialize the worklist
7      initialize empty worklist list W;
8      add colors π(V) in increasing order to W;
9      // work on worklist
10     while W is non-empty do
11         take a color c from W;
12         SplitWithRespectTo(G, π, c, W);
13     return π
```

following:

$$\sum_{i \in 1,\ldots,l} k_i \log k_i \leq n \log n$$

Having removed non-splitting colors, sorting Col_{old} using, e.g., merge sort, can therefore be bounded by $\mathcal{O}(n \log n)$. The reason is that each color that is split, is split into at least 2 new parts: hence, there is at least one new color for each entry of Col_{old}.

We can use a similar argument for sorting the list UniqueDeg. We record the following fact. Assume that for each k_i, $k_i \geq 1$ holds. This implies that for the number of iterations $l \leq n$ holds. Then, it follows that

$$\sum_{i \in 1,\ldots,l} (k_i + 1) \log(k_i + 1) \leq 2n \log 2n = \mathcal{O}(n \log n).$$

Now we note two things: first, all elements of UniqueDeg except for one must correspond to newly introduced colors. Secondly, UniqueDeg contains at least two elements. Thus, we can apply our argument from above and sort it using merge sort in the appropriate runtime.

Differences to other Versions. The version described in [16] mainly differs in two ways: first of all, colors (and other important structures) are maintained as linked lists instead of arrays. While this is convenient to argue about the algorithm, it comes with the well-known downsides for implementations. This results in many parts of the algorithm being very different from the version given here.

Secondly, sorting the degrees(Line 22) is not described as a sorting operation. Instead, a maximal degree for each color is maintained, and potential degrees are simply iterated

Algorithm 10: Refine with respect to a color c in color refinement.

```
1  function SplitWithRespectTo
       Input : ➢ graph G
               ➢ coloring π
               ➢ color c
               ➢ worklist W
      Output : ≺ refines coloring π
               ≺ changes worklist W
   Auxiliary : ✏ reset array Deg of length n
               ✏ reset array Num_Col of length n
               ✏ small integer set Col_old of length n
               ✏ small integer set UniqueDeg of length n
               ✏ array Adj_Col of length n
               ✏ array Deg_Col of length n
2      // count neighbors of c
3      for ( each v ∈ π^{-1}(c) )
4          for ( each neighbor v' ∈ N(v) )
5              x := π(v');
6              if x ∉ Col_old then add x to Col_old ;
7              if Deg[v'] = 0 then
8                  Adj_Col[ x + Num_Col[x]] = v';
9                  Num_Col[x] += 1;
10             Deg[v'] += 1;
11     // remove non-splitting colors and then sort
12     remove x ∈ Col_old where all v, v' ∈ π^{-1}(x) have Deg[v] = Deg[v'];
13     sort Col_old in increasing order ;
14     // split up colors according to neighbor counts
15     for ( each x ∈ Col_old )
16         (n_1, ..., n_l) := Adj_Col[x], ..., Adj_Col[x + Num_Col[x] − 1];
17         for ( each v in (n_1, ..., n_l) )
18             if Deg[v] not in UniqueDeg then
19                 add Deg[v] to UniqueDeg;
20                 Deg_Col[Deg[v]] = 0;
21             Deg_Col[Deg[v]] += 1;
22         sort UniqueDeg in increasing order;
23         s := x + |π^{-1}(x)| − Num_Col[x];
24         for ( each d in UniqueDeg )
25             s' := Deg_Col[d];
26             Deg_Col[d] = s;
27             if c ≠ s and ReportSplit(π, c, s, d) then return;
28             s += s';
29         Rearrange(π, x, (n_1, ..., n_l), Deg, Deg_Col);
30         ManageWorklist(π, W, x, (x_1, ..., x_k));
31         reset UniqueDeg;
32     reset Deg, Num_Col, and Col_old;
```

Algorithm 11: Rearrange and split color class c.

```
1  function Rearrange
       Input: ➢ coloring π
              ➢ color c
              ➢ list of vertices (n_1, ..., n_l)
              ➢ array Deg
              ➢ array Deg_Col
      Output: ≺ refines coloring π
              ≺ manipulates Deg_Col
2      // set all VertexToCol_π to the correct color
3      for ( each v ∈ (n_1, ..., n_l) )
4          d := Deg[v];
5          c' := Deg_Col[d];
6          VertexToCol_π[v] = c';
7          ColToSize_π[c'] = 0;
8      // adjust ColToSize_π appropriately
9      for ( each v ∈ (n_1, ..., n_l) )
10         c' := VertexToCol_π[v];
11         ColToSize_π[c'] += 1;
12         ColToSize_π[c] -= 1;
13     // rearrange Lab_π and VertexToLab_π
14     for ( each v ∈ (n_1, ..., n_l) )
15         d := Deg[v];
16         p := Deg_Col[d];
17         Deg_Col[d] += 1;
18         p' := VertexToLab_π[v];
19         v' := Lab_π[p'];
20         Lab_π[p] := v;
21         Lab_π[p'] := v';
22         VertexToLab_π[v] := p;
23         VertexToLab_π[v'] := p';
```

Algorithm 12: Manage the worklist according to the given split.

```
1  function ManageWorklist
       Input: ➢ coloring π
              ➢ worklist W
              ➢ color x
              ➢ list of colors (x_1, ..., x_k)
      Output: ≺ changes worklist W
2      let i be the smallest i where ∀j ∈ 1, ..., k : |π(x_i)| ≥ |π(x_j)|;
3      add x_1, ..., x_k except x_i to W;
4      if x ∈ W then replace x in W with x_i;
```

from 1 to the maximal degree. So in essence, it describes a bucket sort of the list. This is amortized by the fact that the appropriate number of edges has been seen by the algorithm anyway.

The algorithm described here is inspired by the implementation of Traces. This means that, unsurprisingly, the description here is quite close to the algorithm of Traces. While there are of course differences, most of the maintained data structures are similar, while some additional optimizations of Traces are described in Section 4.3. What is noteworthy however, is that Traces implements the two sorting operations (Line 13 and Line 22) using the quicksort algorithm. This means that, while typically not an issue in practice, the color refinement of Traces indeed does not run in the $\mathcal{O}((n+m)\log n)$ worst-case time, simply due to the sorting operations.

4.2 Worklist Order

Algorithm 9 maintains the classes with respect to which refinements still have to be performed in a worklist W. Note that the algorithm does not fully specify the internals of the worklist. Specifically, it does not state in Line 11 which cell is extracted from the worklist.

The worklist order used in practice does differ between the solvers, but there are common themes. First of all, stack-based worklists are preferred over other approaches [76, 27]. The reasoning is quite simple, and has nothing to do with the asymptotic behavior of the algorithm: colors that are on the top of stack, tend to be colors which have been looked at recently. Thus, the intuition is that these colors have a higher chance of still being cached by the computer than others.

Next, there is a preference for smaller color classes, or more specifically singletons. The reason is explained in detail in Section 4.3.1, where we describe an optimized variant of Algorithm 10 for singleton color classes. The benefit of this optimized singleton variant over the normal split algorithm is quite significant in practice. Traces looks at the first 12 elements of the stack and prefers the first, smallest one of these color classes. saucy maintains two different stacks: one for singletons, one for non-singletons. As long as the singleton stack is non-empty, it is preferred over the non-singleton stack.

Let us now analyze this design choice formally. As mentioned previously, we define an online model, which enables us to compare and analyze different worklist strategies formally. In this model, we prove that no worklist strategy is competitive to all other worklist strategies. We then corroborate the result with a second offline version of the problem.

The following results are based on a model for color refinement as was introduced in [16]. We mention the following technicality: in our implementation, we *refine with respect to a color class* C. In particular, we refine *all other* color classes with respect to the color C. In the lower bound model, we allow algorithms to operate in a more fine-grained manner. In particular, we allow algorithms to refine a union of color classes X with respect to a union of color classes C. This means only the union of color classes X is split according to its neighbor counts in C. Clearly, our color refinement implementation

also operates within this model.

4.2.1 Online Model

Partial Quotient Graphs. For an equitable partition, quotient graphs capture the information of how many neighbors vertices from one class have in another class. They are used in so-called individualization-refinement algorithms as pruning invariants, as was discussed in Section 2.3.4. Typically, information about the quotient graph is computed on the fly during the execution of a color refinement algorithm.

We now introduce the concept of *partial quotient graphs*. These graphs are a tool to formalize the information gathered up to a certain point during the execution of color refinement algorithms. As we cannot precisely say which information an algorithm collects, the quotient graphs give an overapproximation of the available information and model all information that could have possibly been gathered. For the purpose of our lower bounds, overapproximating can only strengthen the conclusions.

The partial quotient graph of a colored graph (G, π) is denoted by $P(G, \pi)$. Quotient graphs are directed and contain self-loops. They include vertex labels l_V as well as edge labels l_E. The vertex set of $P(G, \pi)$ is the set of all sets of colors of (G, π), i.e., $V(P(G, \pi)) \coloneqq 2^{\pi(V(G))}$. A set of colors represents the class that is the union of the respective color classes.

Vertices of the partial quotient graph are labeled with the size of their corresponding set of vertices in G, i.e., for all sets of colors $c \in 2^{\pi(V(G))}$ we define

$$l_V(P(G, \pi))(c) \coloneqq |\pi^{-1}(c)|.$$

The edge set contains all connections between (unions of) color classes that would not cause a split. Thus there is an edge from c_1 to c_2 if $\pi^{-1}(c_2)$ does *not* split $\pi^{-1}(c_1)$. Formally, this means

$$E(P(G, \pi)) \coloneqq \{(c_1, c_2) \mid c_1, c_2 \in 2^{\pi(V(G))}, \forall v, w \in \pi^{-1}(c_1) : \ d_{\pi^{-1}(c_2)}(v) = d_{\pi^{-1}(c_2)}(w)\}.$$

Edges only exist whenever the connection between unions of color classes are regular on one side, so we can label each edge with the corresponding degree, i.e.,

$$l_E(P(G, \pi))((c_1, c_2)) := d_{\pi^{-1}(c_2)}(v),$$

where $v \in \pi^{-1}(c_1)$ is arbitrary.

Let us justify the definition with an example. Suppose we split in a monochromatic graph the class of all vertices with itself. Then the new coloring partitions the vertices precisely by degree. That is, classes contain vertices of the same degree. An algorithm would know this degree, since it has counted the edges incident with each vertex, but it would not know how many neighbors a vertex has within a current color class. In the partial quotient graph, there is an edge from each new color class to the union of all color classes.

The definition of partial quotient graphs contains many more vertices and edges and information on these than would truly be available while executing color refinement. In

Algorithm 13: Corresponding color refinement for an online strategy W.

```
1 function ColorRefinement
      Input: ➢ graph G
             ➢ coloring π
     Output: ≺ refined coloring π
2   create list S containing P(G, π);
3   while π is not equitable do
4       (C, X) := W(S);
5       for each vertex in X count its neighbors in C;
6       split X into X_1, ..., X_k in π, according to neighbor counts;
7       append P(G, π) to S;
8   return π
```

fact, partial quotient graphs grow exponentially in size, since all possible unions of color classes are considered. Common color refinement algorithms clearly gather much less information. Firstly, only connections of classes that are involved in a refinement are actually considered. Secondly, only information about unions of colors that occurred as a color class in a previous step of the refinement is known. Thus, usually color refinement algorithms only uncover a small, polynomial-sized portion of the partial quotient graphs defined above.

However, for our lower bounds, we assume that algorithms have access to the entire partial quotient graph. We show that even if we generously allow such access, the information is not sufficient to derive a strategy with constant competitive ratio. For upper bounds, we only use information of the aforementioned polynomial-sized portion of partial quotient graphs. In fact, the upper bounds are based on a stack-based approach, akin to a simple worklist as used by Algorithm 9.

Model. We now define a model that bases the choice of which color classes to use for the next refinement solely on the information available through partial quotient graphs. Practical implementations such as a queue or a stack are naturally captured by this, but the model even allows for much more powerful choices. The goal is then to prove that no strategy based solely on information of partial quotient graphs is sufficient to make optimal choices.

Let us start by defining the concept of a *strategy* $W : \mathcal{P}^* \to (2^{\mathbb{N}})^2$. A strategy is a function mapping a string of quotient graphs $P = P_1 \cdots P_k \in \mathcal{P}^*$ to two vertices of the last quotient graph $(C, X) \in V(P_k)^2$, that is, two unions of color classes. The string of graphs P denotes all partial quotient graphs observed during execution of the algorithm up to step k. The pair (C, X) denotes the choice of colors with which the algorithm continues in the next step: in step $k+1$, the algorithm refines X with respect to C.

For a strategy W we now define a *corresponding color refinement implementation*. Assume we are working on G and have already refined up to a coloring π_k within k steps. Furthermore, let $P_1, \ldots, P_k$ denote the partial quotient graphs corresponding to the execution. Next, we compute $(C, X) = W(P_1 \cdots P_k)$ and refine X with respect to

C. The algorithm terminates whenever π_k is equitable. A formal definition is given in Algorithm 13. We call W a *valid* strategy if the corresponding color refinement implementation is correct, i.e., if it terminates with an equitable partition in finite time on all finite graphs.

We measure the *cost* of the strategy W, denoted $\text{cost}(W, G)$, in terms of the number of edges that need to be considered to execute the refinements. Specifically, when refining X with respect to C, we charge the algorithm the number of edges connecting X with C. This is the same model as used in [16], which ought to reflect the runtime of efficient implementations. We use the terms cost and time interchangeably.

4.2.2 Graph Gadgets

We want to construct graphs that cause color refinement to behave in particular manners. These graphs are mostly built using three types of graph gadgets, described next.

AND_i gadgets. We recall AND_i gadgets as were already used in Chapter 3. Whenever all pairs of *in-vertices* have been split, a split of two *out-vertices* a_0 and a_1 is induced, but not before. The AND_i gadget is constructed recursively using AND_2 gadgets. Figure 4.1 shows how the AND_3 gadget can be constructed using three AND_2 gadgets.

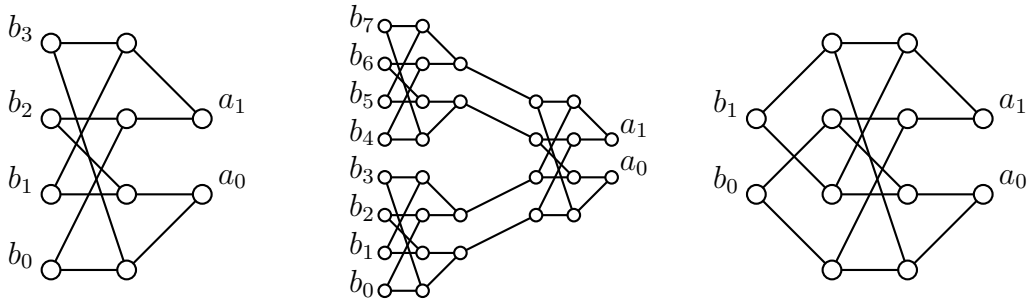
Recall that in an AND_i gadget, all pairs b_{2j}, b_{2j+1} with $j \in \{0, ..., 2^{i-1}\}$ need to be distinguished to induce a split of a_0 and a_1. We should also record a property for the opposite direction: if a_0 and a_1 are distinguished, no split on B should be induced.

Unidirectional gadgets. The *undirectional gadget* used in this section slightly differs from the one defined in Chapter 3. Recall that unidirectional gadgets block the continuation of a split of pairs in one direction but allows it in the opposite direction. Figure 4.1 illustrates the gadget.

The gadget behaves as follows. Consider in-vertices b_0, b_1 and out-vertices a_0 and a_1. Distinguishing b_0 and b_1 should induce a split of a_0 and a_1. However, distinguishing a_0 and a_1 should *not* cause a split of b_0 and b_1. The gadget is obtained through a modification of the AND_2 gadget. We use the fact that a split of out-vertices in AND_2 does not cause a split of the pairs of in-vertices. Therefore, by connecting the in-vertices to new vertices a_0 and a_1, such that the AND_2 gadget is activated by any of the two singletons, we get the desired property.

Concealer gadgets. We conclude our discussion of gadgets with the *concealer gadgets*. Similar to the AND_i gadget, a concealer gadget C_i of level i has 2^i in-vertices B and 2 out-vertices a_0, a_1. Whereas in the AND gadget, *all* input pairs need to be distinguished, the concealer gadget only includes *one* specific pair that causes a split of the out-vertices. We call the pair causing the split of out-vertices the *correct pair*, while all other pairs not causing the split are called *dead end pairs*.

The idea is that the correct pair can not be located easily by color refinement algorithms. Hence, the gadget *conceals* where refinement can be continued.



(a) An AND_2 gadget. (b) An AND_3 gadget. (c) Unidirectional gadget.

Figure 4.1: Basic gadget constructions as used in the online model construction. Vertices labeled with b_i always denote in-vertices, while a_i denotes out-vertices.

To achieve this behavior, the gadget consists of 2^{i-1} unidirectional gadgets and the out-vertices a_0, a_1. We modify all but one of the unidirectional gadgets so that the connection of the in-gate agrees with the one of the out-gate. This causes these gadgets to become dead ends – activating any of these gadgets has no effect on the out-vertices. The last, unmodified unidirectional gadget is the only one that can actually split the out-vertices and is therefore the only correct gadget.

The out-vertices of the entire concealer gadget are then connected to the out-vertices of all the unidirectional gadgets so that activating the correct pair causes a split of the out-vertices. Figure 4.2 shows a concealer gadget C_3.

Since we did not specify which of the pairs is the correct pair, there are several concealer gadgets for each $i \in \mathbb{N}$. Abusing notation we denote all of them by C_i. The concealer gadgets have two crucial properties. First, as long as the correct pair has not been split (and the neighbors of a correct pair have not been split) the partial quotient graphs of two concealer gadgets on the same size are isomorphic. Second, the correct pair can only be split from outside the gadget. We formalize these properties in the following.

Consider two colored concealer gadgets $(C_i, \pi), (C_i', \pi')$ of the same order. Suppose $\{b_s, b_{s+1}\}$ is the correct pair in (C_i, π) and $\{b_t, b_{t+1}\}$ is the correct pair in (C_i', π'). We say the two graphs still *concur* if the colors for the vertices agree (note that the two graphs have the same vertex set) and in both graphs neither the correct pairs nor their neighbors have been split. Specifically, we require that

- the vertex colorings agree, (i.e., $\pi(v) = \pi'(v)$ for every $v \in V(C_i) = V(C_i')$),
- the correct pairs have not been distinguished (i.e., $\pi(b_s) = \pi(b_{s+1})$ and $\pi'(b_t) = \pi'(b_{t+1})$),
- the neighbors of the correct pairs have not been distinguished, i.e., $\pi(v) = \pi(v')$ for all $v, v' \in N_{C_i}(b_s) \cup N_{C_i}(b_{s+1})$ and $\pi(v) = \pi(v')$ for all $v, v' \in N_{C_i'}(b_t) \cup N_{C_i'}(b_{t+1})$. Here, N_G denotes the neighborhood in graph G.

Lemma 61. *Suppose (C_i, π) and (C_i', π') are colored concealer gadgets that concur. Then the graphs have the same partial quotient graphs, i.e., $P(C_i, \pi) = P(C_i', \pi')$.*

Proof. Suppose for a vertex v we want to count the number of neighbors that v has in a union of color classes X. We claim that this number is the same in C_i and C'_i. Indeed, we only need to consider edges incident with v that have one endpoint in $M = \{b_s, b_{s+1}, b_t, b_{t+1}\}$ and one endpoint in $N[M]$ (the neighborhood of M). Let E' be the set of these edges and let E'_v be the set of these edges incident with v.

Note that for each of the four sets $\{b_s, b_{s+1}\}$, $\{b_t, b_{t+1}\}$, $N[\{b_s, b_{s+1}\}]$, and $N[\{b_t, b_{t+1}\}]$ either X contains the set entirely or not at all.

If v is in M, then either all edges of E'_v have an endpoint in X or no such edge does.

Likewise, if v is in $N[M]$, then either all edges of E'_v have an endpoint in X or no such edge does.

Moreover, in either case, whether all such edges are or no such edge is contained does not depend on whether we consider C_i or C'_i.

This implies that the number of edges counted in the refinement (i.e., those incident with v and having an endpoint in X) is the same in C_i and C'_i. □

Lemma 62. *For concealer gadgets (C_i, π) and (C'_i, π') suppose $\pi = \pi'$ so that*

- *vertices in an input pair that is correct in one of the graphs have the same color and*
- *all vertices that are not in an input pair have the same color.*

Then (C_i, π) and (C'_i, π') concur. After an arbitrary sequence of splits to both graphs the resulting graphs still concur and neither correct input pairs nor the out pair are split.

Proof. This follows by induction on the number of steps observing that the functionality of the unidirectional gadget ensures that the output pair is never split, and thus vertices inside correct gadgets are never split. □

The two lemmas show that unless a correct pair is split, the gadgets always concur and an algorithm in the online model will have to perform splits consistently on both graphs. Moreover, the output pair is never split.

Intuitively this means that in the online model, an algorithm can only guess which pair is the correct pair. Therefore, when faced with a concealer gadget, the algorithm potentially has to try all input pairs.

4.2.3 Competitive Ratio in Online Model

We prove the non-existence of a c-competitive strategy in the online model. In particular, in this section, we prove the following theorem:

Theorem 63. *For every strategy W of the online model, there is an infinite family of graphs G_k ($k \in \mathbb{N}$) such that $\mathrm{cost}(W, G_k) \in \Omega(\mathrm{opt}(G_k) \cdot \log(\mathrm{opt}(G_k)))$, where $\mathrm{opt}(G_k) \in \Theta(|G_k|)$ is the minimal cost of a strategy on G_k.*

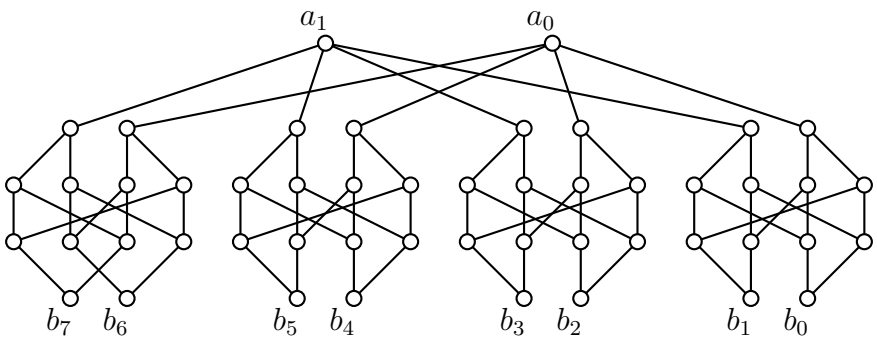


Figure 4.2: A concealer gadget C_3. Vertices b_6, b_7 form the correct pair; other pairs are dead ends.

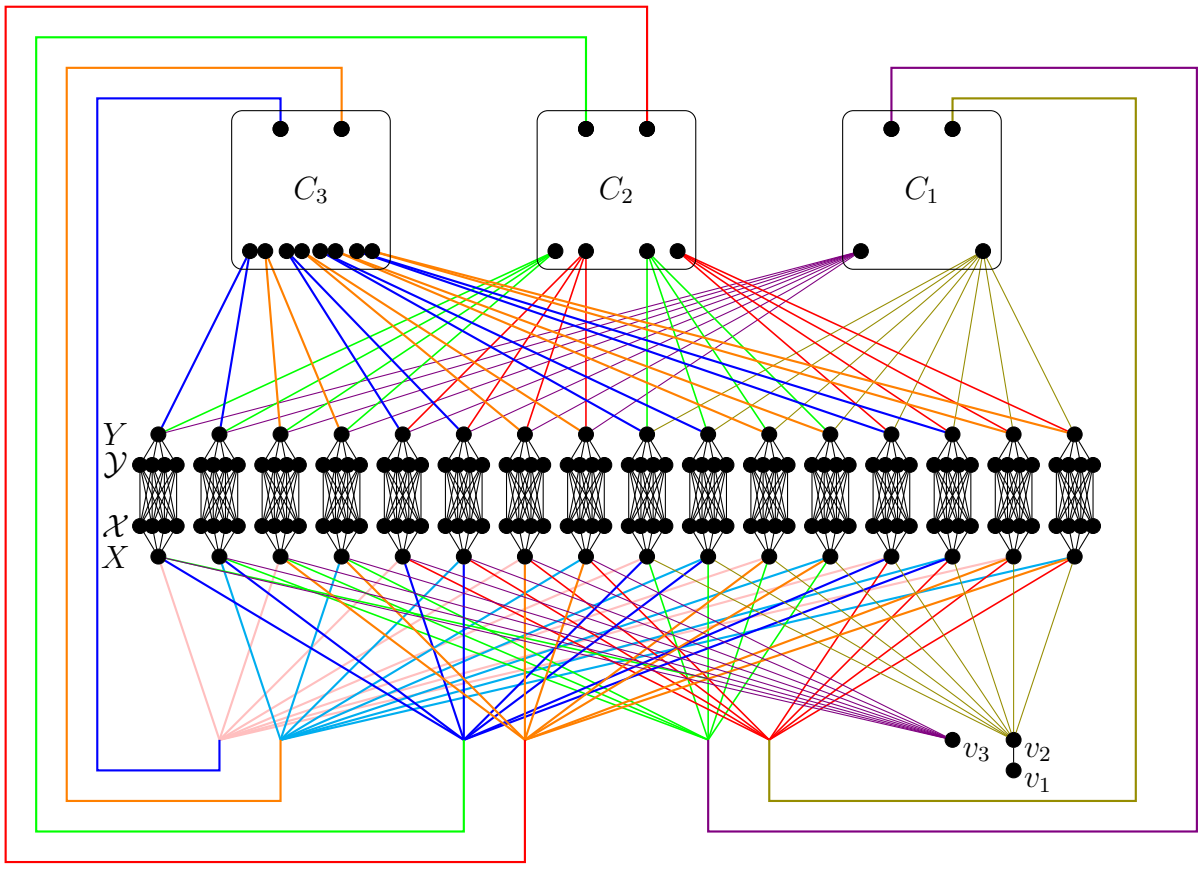


Figure 4.3: A concealer graph from the class $\mathcal{G}_4$.

The theorem implies that the information provided by partial quotient graphs is not sufficient to make competitive let alone optimal choices in color refinement algorithms.

Towards this goal, we first define the class of *concealer graphs*, which we denote with $\mathcal{G}_k$ ($k \in \mathbb{N}$). Concealer graphs resemble the graphs of the lower bound construction in [16] closely. Essentially, we swap out AND_i gadgets in the original construction for concealer gadgets C_i. A concealer graph of $\mathcal{G}_4$ is illustrated in Figure 4.3.

The main idea is that we can then speed-up or slow-down particular strategies by changing the position of the correct pairs within the concealer gadgets. This forces one strategy to extensively search for the correct pairs, while another strategy finds them immediately.

In the rest of this section we provide formal arguments for the above claims. We start with a precise description of concealer graphs. Then, we show that for every concealer graph there exists a fast strategy. Contrarily, we then provide a slow concealer graph for every strategy. Together these two statements prove Theorem 63.

Concealer Graphs. The first ingredient for the concealer graphs is a "splitting scheme" that results in the worst-case running time of $\Omega(m \log(n))$. Consider a vertex set of size $n = 2^k$, on which the following refinements are performed. First, we split the set in halves, then quarters, then eighths and so on, until all vertices have their own distinct color. This gives us $\log(n)$ rounds of refinements, each with a cost of $\Omega(n)$. This results in total costs of $\Omega(n \log(n))$. By ensuring that sufficiently many edges are involved, the running time can be increased to $\Omega(m \log(n))$.

Concealer graphs can be used to cause the splitting scheme just described. The graphs contain *middle layers* $(X, \mathcal{X}, \mathcal{Y}, Y)$ (see Figure 4.3) in which the splitting scheme can be forced. The graph is constructed in a way such that splitting Y into halves, quarters, eighths and so on, causes the next halving refinement on X. The edge colors in Figure 4.3 indicate the splitting scheme. While the halves (yellow and purple) of Y lead to a split of X into quarters (red and green), the quarters of Y lead to eighths (blue and orange) of X and so on. By initially splitting X in halves, any color refinement algorithm needs to cycle through these layers until X is fully discrete.

The core idea of the general lower bound construction in [16] is that the AND_i gadget enforces refinements with respect to *every* block of level i, which in turn ensures costs of $2^k \cdot k^2 \in \Omega(m)$ for every level.

We modify the construction to suit our purposes as follows. In the concealer graphs, we swap for each i the AND_i gadget for a concealer gadget C_i. On a particular graph, the worst-case behavior is therefore not enforced for all refinement strategies anymore. However, a deterministic online algorithm cannot choose for *all* possible concealer gadgets the correct pair in level i to allow it to continue with level $i+1$. Hence, an adversary can construct a graph that makes a specific color refinement slow, while keeping a "shortcut" for other algorithms that choose the correct pair directly.

We now formally define the class $\mathcal{G}_k$ of concealer graphs. Note that for every $k \in \mathbb{N}$, we define a set of graphs $\mathcal{G}_k$. Essentially, we describe a graph $G_k \in \mathcal{G}_k$ based on concealer gadgets, and the set $\mathcal{G}_k$ then simply consists of all possible instantiations (i.e., positions of the correct pairs) for the included concealer gadgets.

At its core, a graph $G_k \in \mathcal{G}_k$ consists of the four middle layers of vertices $(X, \mathcal{X}, \mathcal{Y}, Y)$, that are interconnected using additional gadgets. Formally, the vertex set of G_k includes

$$\begin{aligned} X &\coloneqq \{x_0, \dots, x_{2^k-1}\}, \\ \mathcal{X} &\coloneqq \{x_i^j \mid 0 \leq i < 2^k, 0 \leq j < k\}, \\ \mathcal{Y} &\coloneqq \{y_i^j \mid 0 \leq i < 2^k, 0 \leq j < k\}, \\ Y &\coloneqq \{y_0, \dots, y_{2^k-1}\}, \end{aligned}$$

a simple starting gadget induced by only three vertices v_1, v_2, v_3 and $k-1$ concealer gadgets. For $0 \leq l \leq k$ and $0 \leq q \leq 2^l - 1$ let

$$\mathcal{B}_q^l \coloneqq \{q2^{k-l}, \dots, (q+1)2^{k-l} - 1\}$$

be the q-th binary block of level l. We use this notation on all sets of size 2^k for some $k \in \mathbb{N}$.

Every x_i is connected to a corresponding y_i via a complete bipartite graph of size k consisting of vertices in $\mathcal{X}$ and $\mathcal{Y}$ (see Figure 4.3). Formally, each x_i is connected to all x_i^j, y_i to all y_i^j and x_i^j to all $y_i^{j'}$. For each level $l \in \{1, \dots, k-1\}$, the i-th binary block of level l is connected to the i-th in-vertex of the l-th concealer gadget. Furthermore, for each gadget C_l, we connect a_0 to all X_i^l with i even and a_1 to all X_i^l with i odd. The starting construction splits X into the blocks X_0^0 and X_1^0. We refer to the i-th in-vertex of the l-th concealer gadget as b_i^l and to the i-th out-vertex as a_i^l.

Let us generally consider how a refinement strategy has to operate on G_k. The algorithm starts with the monochromatic coloring of G_k. The first refinement always distinguishes vertices by their degree, meaning we get the individualized starting gadget $\{v_1\}, \{v_2\}, \{v_3\}$, the distinct layers in the middle $X, \mathcal{X} \cup \mathcal{Y}, Y$, the in- and out-vertices of the concealer gadgets

$$\bigcup_{l \in \{1,\dots,k-1\}} \{b_i^l, a_j^l \mid i \in \{0, \dots, 2^l\}, j \in \{0, 1\}\},$$

and the union of the inner vertices of the concealer gadgets. Next the middle layers are split in half. From this point onwards the splits that are possible depend on finding the correct pair in the gadgets. This can lead to fast or slow refinements, as discussed next.

A Fast Strategy for Every Concealer Graph. We now show that for every *fixed* concealer graph $G_k \in \mathcal{G}_k$ we can define a linear time strategy. We show this by providing an appropriate sequence of refinements.

For each concealer gadget C_l in G_k, let $b_{i_l}^l, b_{i_l+1}^l$ be the correct pair. Now consider an online refinement strategy on such a graph. After the first (and fixed) refinement, we refine X with respect to $\{v_2\}$ or $\{v_3\}$. We choose one half of $X_{i_1}^1$ for the next refinement and then $\mathcal{X}_{i_1}^1$, $\mathcal{Y}_{i_1}^1$ and $Y_{i_1}^1$ while propagating the split through the middle layers. The important property is that $Y_{i_1}^1$ always splits the correct pair of the next concealer gadget. The concealer gadget then in turn splits X into quarters. Now, we continue with the

quarters $X^2_{i_2}$, $\mathcal{X}^2_{i_2}$, $\mathcal{Y}^2_{i_2}$ and $Y^2_{i_2}$, such that the second concealer gadget is activated. This splits X in eighths.

We now repeat this scheme, such that for each level we only propagate the blocks corresponding to correct pairs through the layers and immediately continue with the next level after activating the concealer gadget. When X is discrete, we get the equitable coloring by refining with respect to each level k block of X, $\mathcal{X}$, $\mathcal{Y}$ and Y.

Now consider the cost of this strategy. While cycling through the layers, the most expensive refinements are those with respect to the blocks of $\mathcal{X}$ and $\mathcal{Y}$. On level l, they have cost $2^{k-l} \cdot k^2$, which means the total cost for all levels is $2^k \cdot k^2 = \Theta(m)$. Once X is discrete the cost of the final refinements of $\mathcal{X}$, $\mathcal{Y}$ and Y is also in $\Theta(m)$.

Overall, the cost for an optimal solution for G_k is linear, i.e., $\text{opt}(G_k) \in \Theta(m)$. Note that since refinement is always continued with color classes that have just been created, the scheme actually follows a depth-first approach and could be implemented using a stack (barring a particular order in which newly created color classes are pushed to the stack).

A Slow Concealer Graph for Every Strategy. For a *fixed* strategy W, we now provide an infinite family of concealer graphs G_k on which this strategy is slow, i.e., incurs super-linear cost. The family is constructed by choosing for every $k \in \mathbb{N}$ one specific concealer graph $G_k \in \mathcal{G}_k$.

We start with an arbitrary graph $G_k \in \mathcal{G}_k$. We run W on G_k and observe which color classes are split within the concealer gadgets. Say we are looking at concealer gadget C_i. If W distinguishes the correct pair in G_k, but there are still dead ends that have not been distinguished, then we replace G_k by the graph $G'_k \in \mathcal{G}_k$ obtained from G_k by replacing the gadget C_i with another one so that a dead end not yet investigated becomes the correct pair. Due to Lemma 61 and Lemma 62 we know that up until the point where W finds the correct pair in C_i for graph G_k, the strategy W performs the same sequence of splits when executed on G'_k as on G_k. Thus, by doing these transformations exhaustively, we ensure W distinguishes all correct pairs in all the concealer gadgets last. This causes $2^k \cdot k^2$ cost per level and hence $2^k \cdot k^3 = \Theta(m \log(n))$ total cost.

Since the optimal solution for fixed G_k only has linear cost, we in turn get that

$$\text{cost}(W, G_k) \in \Omega(\text{opt}(G_k) \cdot \log(\text{opt}(G_k))).$$

We now argue this in more detail. Let us first discuss some general behavior of color refinement on concealer graphs. There are two core properties that hold for every color refinement algorithm. The first one is that in each level l, we split X completely into the blocks of this level, $X^l_0, ..., X^l_{2^l}$. The other layers can only be split by the blocks of X, so we know that their partitions are always coarser than the one of X. The second property is that out-vertices of the level l concealer gadget have to be distinguished to partition X into the blocks of level $l+1$. Thus, also the correct input pair in this gadget has to be split.

For a coloring α we denote by π_α the partition induced by the coloring. The notation $\pi_\alpha[X]$ indicates the restriction of the partition to a set X and we use $\pi_\alpha \preceq \pi_{\alpha'}$ to

indicate that the former partition is at least as fine as the latter. Abusing notation we compare partitions of the layers of the graphs, as they are related by direct connections.

Lemma 64. *Let α_i denote the coloring after i steps of color refinement. For any α_i with $i \geq 0$ there is a number n_A, such that*

- *for all $j \leq n_A$: $\alpha_i(a_0^j) \neq \alpha_i(a_1^j)$, and*
- *for all $j > n_A$: $\alpha_i(a_0^j) = \alpha_i(a_1^j)$.*

In addition, we have either

- $\pi_{\alpha_i}[X] = \{X_q^{n_A+1} \mid 0 \leq q \leq 2^{n_A+1} - 1\}$ *or*
- $\pi_{\alpha_i}[X] = \{X_q^{n_A} \mid 0 \leq q \leq 2^{n_A} - 1\}$.

Furthermore, it holds that $\pi_{\alpha_i}[X] \preceq \pi_{\alpha_i}[\mathcal{X}], \pi_{\alpha_i}[\mathcal{Y}], \pi_{\alpha_i}[Y]$.

Proof. For $i = 0$ this is obviously true with $n_A = 0$.

For later iterations, we consider the possible splits. The start vertices v_1, v_2 and v_3 are not able split X any further after the first iteration. Any part of X in π_{α_i} can split $\mathcal{X}$, but not A, since a_0^j and a_1^j are equally connected to all $X_q^{n_A}$ and $X_q^{n_A-1}$ for all $j > n_A$.

Due to the simple one-to-k connection from X to $\mathcal{X}$ and because X is already finer, none of these splits makes $\mathcal{X}$ finer than X. Since $\mathcal{X}$ is coarser than X, $\pi_{\alpha_i}[X]$ will not be changed by subsets of $\mathcal{X}$. With the same argument, $\mathcal{X}$ does not make $\mathcal{Y}$ coarser than X and vice versa. The same holds for $\mathcal{Y}$ and Y.

Now consider the case that $\pi_{\alpha_i}[X] = \{X_q^{n_A} \mid 0 \leq q \leq 2^{n_A} - 1\}$. Since Y is coarser, no part of Y can yield an activated gadget C_{n_A}. Also, any other splits of concealer gadgets (which do not split input pairs) do not change the claimed property. Only distinguished out-vertices of the concealer gadgets can cause splits of X, which are all the a_0^j, a_1^j with $j \leq n_A$. Out of those, only $a_0^{n_A}$ and $a_1^{n_A}$ can further split X into the blocks of level n_A+1, not changing the claimed property. The other levels can only split X into blocks of lower level, which has already been done.

Otherwise, i.e., if $\pi_{\alpha_i}[X] = \{X_q^{n_A+1} \mid 0 \leq q \leq 2^{n_A+1} - 1\}$, there can be a part Y' of Y in $\pi_{\alpha_i}[Y]$ with $Y' = Y_q^{n_A+1}$ for some q. Then Y' can activate C_{n_A+1}, which leads to a split of $a_0^{n_A+1}$ from $a_1^{n_A+1}$. This will increase n_A by 1 (since X was partitioned into blocks of level $n_A + 1$, the claimed property is preserved). A split of X from $a_0^{n_A}$ or $a_1^{n_A}$ cannot happen in this case.

A split from the in-vertices of the concealer gadgets to Y will also never make Y finer than X, since C_{n_A}, the gadget which can split Y into the finest partitions under all the pairs in A, is connected to the n_A-th level of Y, and X has already been split into the blocks of level at least n_A. □

With this lemma, we can define an adversary that constructs a graph such that a specific strategy shows worst-case behavior. It should be noted that the lemma can also be stated with a_0^j/a_1^j exchanged by $b_{i_j}^j/b_{i_j+1}^j$, since the split of the former is directly

dependent on the split of the latter. The lemma is stated in this way so that we can reuse it at a later point.

Let $\mathcal{A}$ be the corresponding color refinement to some strategy W. We construct an infinite family of graphs on which $\mathcal{A}$ has costs of $\Omega(m \log(n))$. In the family there is for each $k \in \mathbb{N}$ a graph $G_k \in \mathcal{G}_k$. We start with a concealer graph $G \in \mathcal{G}_k$ and then successively specify the position of the correct pairs.

Let π_t be the partitions of G that $\mathcal{A}$ produces in step t. Consider an arbitrary step t in the execution of $\mathcal{A}$, where

$$\pi_t[X] = \{X_q^{n_A+1} \mid 0 \leq q \leq 2^{n_A+1} - 1\}$$

for the unique n_A from the previous lemma, but the correct pair $b_{i_{n_A}}^{n_A}, b_{i_{n_A}+1}^{n_A}$ of the level n_A concealer gadget has not been distinguished. We know that $\mathcal{A}$ needs to split $b_{i_{n_A}}^{n_A}, b_{i_{n_A}+1}^{n_A}$ to continue to the next level. Let t_{next} be the largest t' such that

$$\pi_{t'}[\{b_0^{n_A}, \ldots, b_{2^{n_A}-1}^{n_A}\}] = \pi_t[\{b_0^{n_A}, \ldots, b_{2^{n_A}-1}^{n_A}\}],$$

i.e., the next point in time where the in-vertices of the current concealer gadget are split. We assume w.l.o.g. that this is a split of an input pair. Let $b_{i_{\text{next}}}^{n_A}, b_{i_{\text{next}}+1}^{n_A}$ be the in-vertex pair which is distinguished at t_{next}. An adversary can choose the concealer gadget of level n_A such that $b_{i_{\text{next}}}^{n_A}, b_{i_{\text{next}}+1}^{n_A}$ is a dead end pair.

Due to Lemma 61 and Lemma 62, the behavior of $\mathcal{A}$ until step t_{next} stays the same, no matter what concealer gadget is used in G. With the previous lemma, we know that after distinguishing $b_{i_{\text{next}}}^{n_A}, b_{i_{\text{next}}+1}^{n_A}$, we are in the same situation as before, i.e. X is partitioned into the blocks of level n_A, but the correct pair $b_{i_{n_A}}^{n_A}, b_{i_{n_A}+1}^{n_A}$ has not been split (thus, another split of an input pair is needed to increase n_A).

If at step t we are in the case that

$$\pi_t[X] = \{X_q^{n_A} \mid 0 \leq q \leq 2^{n_A} - 1\},$$

no important splits happen, since for all levels $l \leq n_A$, every input pair of C_l has already been split and for all $l \geq n_A$, no splits of input pairs are possible.

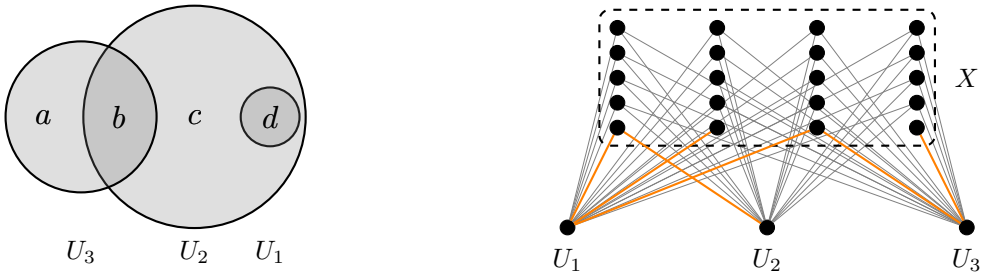
Thus, we can repeat these changes as often as necessary such that the correct pair is only split after all the dead end pairs have been split. By doing this for each level, we get a graph on which $\mathcal{A}$ has cost $\Omega(m \log(n))$.

4.2.4 Competitive Ratio in Offline Model

Complementing our previous results, we now provide an approximation hardness result for computing optimal color refinement strategies. We begin by defining the optimal refinement worklist problem:

Problem 65 (Refinement Worklist Problem)**.** *Given a colored graph (G, π), compute a minimal cost sequence of pairs of color classes $W = (C_1, X_1), \ldots, (C_t, X_t)$ such that:*

1. *Refining with respect to W results in the stable coloring π^∞.*



(a) Set cover instance.

(b) Result of the reduction.

Figure 4.4: Reduction of the set cover instance $S = \{a, b, c, d\}$ and $\mathcal{U} = \{\{d\}, \{b, c, d\}, \{a, b\}\}$. Orange lines indicate connections to elements of S, all other edges are connections to dummy elements.

2. *For all prefixes $(C_1, X_1), \ldots, (C_s, X_s)$, the partial quotient graph obtained after refining C_i w.r.t. X_i for $i = 1, \ldots, s-1$ contains C_s and X_x (as unions of color classes).*

The cost of a sequence W is the sum of the costs for refining with respect to all $(C_i, X_i) \in W$.

The approximation hardness result is based on a reduction from the set cover problem. The set cover problem takes a finite universe S and a set of subsets of S, i.e., $\mathcal{U} \subseteq 2^S$. The decision variant then asks whether there exists a selection of k subsets in $\mathcal{U}$ whose union equals S. For simplicity, we assume $\bigcup_{U \in \mathcal{U}} U = S$. Set cover is well-known to be NP-complete.

The optimization variant requires a minimal selection of subsets that cover S, i.e., a solution that minimizes k. This problem is known to be NP-hard. More specifically, it is known that unless P = NP, polynomial time algorithms can only reach an approximation factor of $\Omega(\log(n))$ [96].

Theorem 66. *Unless P = NP, polynomial time algorithms may only reach an approximation factor of $\Omega(\log(n))$ for the optimal refinement problem.*

Proof. We reduce the optimization variant of the set cover problem to the refinement worklist problem. More specifically, we reduce it in a manner which allows control of the parameters, so that the approximation hardness result of set cover immediately transfers to refinement worklists. The reduction is illustrated in Figure 4.4.

Given a set cover instance $(S, \mathcal{U})$ we define a related colored graph (G, π). We create one large color class X containing all elements of the universe S, as well as n^2 *dummy elements* (where n is the size of the set cover instance). Hence, the size of X is $n^2 + |S|$.

We add a singleton color class for each subset $U \in \mathcal{U}$, i.e., we add vertex U with color U. We connect the vertex U with all vertices of X *except* for the elements that are contained in U. Formally, we define the edges

$$E(G) \coloneqq \{\{U, x\} \mid x \in X \wedge x \notin U\}.$$

Note that U has $n^2 + |S| - |U|$ connections to X.

In the constructed graph, all elements of the universe are eventually distinguished from the dummy elements in X. Refining X with respect to X is not productive, since there are no edges present and no splits occur. The only way to distinguish elements of X is to refine X with respect to an element of $\mathcal{U}$. Doing so always distinguishes all the elements contained in $U \in \mathcal{U}$ from the dummy elements and other remaining elements of X. Overall, we need to refine X with a subset of $\mathcal{U}$ that forms a set cover of S.

After that, assuming all elements of S have been distinguished from the dummy elements, it might be possible to split the resulting classes further through their connections to $\mathcal{U}$. However, the total cost for these further refinements is bounded by $c \cdot n^2$ for some fixed constant c.

The cost for refining X with respect to U is n^2+m, where m is the number of remaining elements of S in X *after* the elements of U have been removed. Since we need to choose at most $|S|$ subsets in a reasonable solution (otherwise we could remove redundant elements from the solution), and each time X gets smaller by at least one element, the cost incurred by m over all subsets is at most $|S|^2 \leq n^2$. Ignoring the cost of m, we get that each subset incurs additional cost of n^2 through the dummy elements.

Hence, the final cost is at most

$$c \cdot n^2 + (N_U + 1) \cdot n^2 = (N_U + c + 1)n^2$$

and at least $N_U n^2$, where N_U is the number of chosen subsets.

We finish our arguments with a proof by contradiction. Assume there is a polynomial time algorithm with an approximation factor in $o(\log(n))$. Given a set cover instance $(S,\mathcal{U})$, we apply the polynomial time reduction stated above. Assume now we get an approximate solution with cost $x \cdot n^2$. We know that this implies a set cover solution with cost at most x.

The optimal set cover solution with cost x' would imply a worklist solution with cost at most $(x'+c)n^2$ (for a fixed c). Hence, we know that the worklist solution also approximates the optimal solution of the original set cover instance with a factor in $o(\log(n))$.

The set cover instance has a size in the 3rd root of the size of the refinement worklist problem. But since $o(\log(n)) = o(\log(\sqrt[3]{n}))$, we get a contradiction to the approximation hardness result of set cover. □

4.3 Split Algorithms

Next, we describe the different split algorithms, to be used within Algorithm 9. The motivation for these different split algorithms is of practical nature, and they do not result in a better theoretical asymptotic runtime. As discussed previously, the split algorithms often contain the hot spot for the entire IR algorithm and hence, a high level of optimization is important. We will describe the different routines, and discuss cases in which they can be applied without detriment to the theoretical runtime. The different split algorithms are reverse-engineered from the implementation of Traces [76].

Algorithm 14: Split with respect to a singleton color c.

```
1  function SplitWithRespectToSingleton
       Input:  ➢ graph G
               ➢ coloring π
               ➢ color c
               ➢ worklist W
       Output: ≺ refines coloring π
               ≺ changes worklist W
       Auxiliary: ✏ reset array Num_Col of length n
                  ✏ small integer set Col_old of length n
                  ✏ array Adj_Col of length n
2      // count neighbors of c
3      v := π^{-1}(c);
4      for ( each neighbor v' ∈ N(v) )
5          x := π(v');
6          if Num_Col[x] = 0 then add x to Col_old ;
7          Adj_Col[x + Num_Col[x]] = v' Num_Col[x] += 1;
8      // remove non-splitting colors and then sort
9      remove x ∈ Col_old where all v, v' ∈ π^{-1}(x) have Deg[v] = Deg[v'];
10     sort Col_old in increasing order ;
11     // split up colors according to neighbor counts
12     for ( each x ∈ Col_old )
13         n_1, ..., n_l := Adj_Col[x], ..., Adj_Col[x + Num_Col[x] − 1];
14         x' := x + |π(x)| − Num_Col[c];
15         if x ≠ s then
16             if ReportSplit(π, c, s, 1) then return;
17         RearrangeSingleton(π, x, (n_1, ..., n_l));
18         ManageWorklist(π, W, x, (x, x'));
19     reset Num_Col, and Col_old;
```

The descriptions here are included purely for the sake of completeness and are meant as a reference for how efficient color refinement is implemented. They have no further bearing on other results of this thesis.

4.3.1 Singleton Split

A crucial optimization employed by state-of-the-art solvers is a split routine specifically tailored to splitting with respect to a singleton color class $C = \{v\}$. A description of this optimization can be found in Algorithm 14 and Algorithm 15.

We remark on the differences to Algorithm 10 and Algorithm 11. The crucial observation is that any vertex of the graph can either be connected or not connected to v. This means that for every vertex there is only two potential neighbour counts: connected or

Algorithm 15: Rearrange and split color class c.

```
1  function RearrangeSingleton
       Input: ➢ coloring π
              ➢ color c
              ➢ list of vertices (n_1, ..., n_l)
      Output: ≺ refines coloring π
2      // set ColToSize_π
3      c' := x + |π(x)| − l;
4      ColToSize_π[c] := |π(x)| − l;
5      ColToSize_π[c'] := l;
6      // rearrange Lab_π and VertexToLab_π
7      p = c';
8      for ( each v ∈ (n_1, ..., n_l) )
9          VertexToCol_π[v] = c';
10         p' := VertexToLab_π[v];
11         v' := Lab_π[p'];
12         Lab_π[p] := v;
13         Lab_π[p'] := v';
14         VertexToLab_π[v] := p;
15         VertexToLab_π[v'] := p';
16         p += 1;
```

disconnected. This removes the need for keeping track of the Deg array, or any part of the routine to determine the colors of the split vertices.

While there seems to be no asymptotic advantage to using Algorithm 14 and Algorithm 15, it should still be quite apparent that the algorithms have a practical advantage in terms of using much fewer auxiliary arrays, and performing less overall operations. Specifically in the individualization-refinement context singletons occur often: every individualization creates a singleton, and the subsequent refinement starts only with this singleton on the worklist.

4.3.2 Dense Split

We simplify the split algorithm if the graph is *dense*. In particular, we observe that if a graph is dense, i.e., has asymptotically more edges than vertices, then the splitting routine will spent most time iterating over the edges adjacent to the color of consideration. The optimization for dense graphs therefore aims to reduce the number of operations that are performed on each incident edge.

Description of Algorithm 16. Compared to Algorithm 10, the algorithm does *not* keep track of the vertices adjacent to the input color c in the first two for-loops of Line 3 and Line 4. This severely reduces the number of operations performed in these loops.

When splitting the color classes however, the algorithm can not efficiently access vertices which have non-trivial degree, and must therefore consider all the vertices of connected color classes (Line 13).

Runtime of Algorithm 16. Clearly, the additional cost incurred by considering every vertex of connected colors is bounded by $\mathcal{O}(n)$. Let us fix a constant $\alpha \in \mathbb{Q}$ with $0 < \alpha \leq 1$, such that we only use Algorithm 16 in case a vertex $v \in \pi^{-1}(c)$ has degree of $\deg(v) > \alpha n$. Under this assumption, our asymptotic worst-case runtime analysis of Algorithm 10 remains unchanged.

4.3.3 Very Dense Split

We can go even further by not keeping track of which colors are adjacent to the color of interest.

Description of Algorithm 17. Compared to Algorithm 16, the algorithm now further does not even keep track of the *colors* adjacent to the input color c in the first two for-loops of Line 3 and Line 4. We only update the degree for each connected vertex.

When splitting the color classes, the algorithm now has to iterate over all color classes of π (see Line 8). Interestingly, this removes the need for *sorting* connected color classes: in a sense, we are replacing the sorting operation with a bucket sort.

Runtime of Algorithm 17. We can use the same argument as for Algorithm 16: for a fixed a constant $\alpha \in \mathbb{Q}$ with $0 < \alpha \leq 1$, we only use Algorithm 17 in case a vertex $v \in \pi^{-1}(c)$ has degree of $\deg(v) > \alpha n$.

Interestingly, the algorithm can be advantageous, even in a theoretical sense: assuming a split divides k many color classes, in case $k \log k > n$, replacing the sorting operation by a simple iteration of color classes can become asymptotically faster.

4.4 Various Optimizations in the IR Context

We now describe further techniques and optimizations, which are in particular important in the context of IR algorithms.

4.4.1 Individualization

Of course, we need to be able to do an individualization. The routine is quite simple and is described in Algorithm 18. It simply rearranges the coloring to place v into its own singleton color class. The routine runs in time $\mathcal{O}(1)$.

By alternating Algorithm 9 and Algorithm 18, we can now compute CRef, i.e., a node in an IR tree. Note that after an individualization, the worklist of Algorithm 9 can be initialized to only contain the singleton color class $\pi(v)$ of the individualized vertex v. Essentially, the split incurred by individualization can be treated like any other color class

Algorithm 16: Split with respect to a color c if the graph is *dense*.

```
1  function SplitWithRespectToDense
       Input:  ➢ graph G
               ➢ coloring π
               ➢ color c
               ➢ worklist W
       Output: ≺ refines coloring π
               ≺ changes worklist W
       Auxiliary: ✏ reset array Deg of length n
                  ✏ small integer set Col_old of length n
                  ✏ small integer set UniqueDeg of length n
                  ✏ array Deg_Col of length n
2      // count neighbors of c
3      for ( each v ∈ π^{-1}(c) )
4          for ( each neighbor v' ∈ N(v) )
5              x := π(v');
6              if x ∉ Col_old then add x to Col_old ;
7              Deg[v'] += 1;
8      // remove non-splitting colors and then sort
9      remove x ∈ Col_old where all v, v' ∈ π^{-1}(x) have Deg[v] = Deg[v'];
10     sort Col_old in increasing order ;
11     // split up colors according to neighbor counts
12     for ( each x ∈ Col_old )
13         n_1, ..., n_l := π^{-1}(x);
14         for ( each v in n_1, ..., n_l )
15             if Deg[v] not in UniqueDeg then
16                 add Deg[v] to UniqueDeg;
17                 Deg_Col[Deg[v]] = 0;
18             Deg_Col[Deg[v]] += 1;
19         sort UniqueDeg in increasing order;
20         s := x;
21         for ( each d in UniqueDeg )
22             s' := Deg_Col[d];
23             Deg_Col[d] = s;
24             if c ≠ s then
25                 if ReportSplit(π, c, s, d) then return;
26             s += s';
27         Rearrange(π, x, (n_1, ..., n_l), Deg, Deg_Col);
28         ManageWorklist(π, W, x, (x_1, ..., x_k));
29         reset UniqueDeg;
30     reset Deg and Col_old;
```

Algorithm 17: Split with respect to a color c if the graph is *very dense*.

```
1  function SplitWithRespectToVeryDense
       Input: ➢ graph G
              ➢ coloring π
              ➢ color c
              ➢ worklist W
      Output: ≺ refines coloring π
              ≺ changes worklist W
   Auxiliary: ✏ reset array Deg of length n
              ✏ small integer set UniqueDeg of length n
              ✏ array Deg_Col of length n
2      // count neighbors of c
3      for ( each v ∈ π^{-1}(c) )
4          for ( each neighbor v' ∈ N(v) )
5              x := π(v');
6              Deg[v'] += 1;
7      // split up colors according to neighbor counts
8      for ( each x ∈ π(V(G)) )
9          n_1, ..., n_l := π^{-1}(x);
10         for ( each v in n_1, ..., n_l )
11             u := 0;
12             if Deg[v] not in UniqueDeg then
13                 add Deg[v] to UniqueDeg;
14                 Deg_Col[Deg[v]] = 0;
15                 u += 1;
16             if u = 1 then continue;
17             Deg_Col[Deg[v]] += 1;
18         sort UniqueDeg in increasing order;
19         s := x;
20         for ( each d in UniqueDeg )
21             s' := Deg_Col[d];
22             Deg_Col[d] = s;
23             if c ≠ s then
24                 if ReportSplit(π, c, s, d) then return;
25             s += s';
26         Rearrange(π, x, (n_1, ..., n_l), Deg, Deg_Col);
27         ManageWorklist(π, W, x, (x_1, ..., x_k));
28         reset UniqueDeg;
29     reset Deg;
```

Algorithm 18: Individualize a vertex in a coloring.

```
1  function Individualize
       Input: ➢ coloring π
              ➢ vertex v
      Output: ≺ refines coloring π
2      // retrieve original color, calculate new color
3      c := VertexToCol_π[v];
4      c' := VertexToCol_π[c] + ColToSize_π[c] − 1;
5      VertexToCol_π[v] := c';
6      // adjust color class sizes
7      ColToSize_π[c] -= 1;
8      ColToSize_π[c'] = 1;
9      // adjust Lab and VertexToLab accordingly
10     p := c';
11     p' := VertexToLab_π[v];
12     v' := Lab_π[p'];
13     Lab_π[p] := v;
14     Lab_π[p'] := v';
15     VertexToLab_π[v] := p;
16     VertexToLab_π[v'] := p';
```

split, and thus we may omit the other fragment of the individualized color class. Note that the cell selector guarantees that for each individualized vertex v, $|\pi^{-1}(\pi(v))| \geq 2$ holds (see Section 2.3). In turn, it is easy to see that any node of the IR tree using color refinement can be computed in time $\mathcal{O}((n+m)\log n)$.

4.4.2 Early Out Opportunities

We describe two ways in which the algorithm may terminate early.

Discrete Coloring. The algorithm may terminate whenever the number of cells in π equals the number of vertices of G, i.e., $|\pi| = |V(G)|$. In this case, it is easy to see that no further refinements can be made. In practice, this is a crucial optimization: anytime a leaf of the IR tree is reached, the remainder of the worklist can simply be ignored.

Note that the number of cells in π can very easily be maintained. Assuming we know the number of cells initially, we can maintain the number very easily using a `ReportSplit` function: whenever there is a split, we increase the number of cells by 1.

Trace Invariant. As discussed in Section 2.3, we want to be able to compute a *node invariant* in the IR tree. In particular, the node invariants we compute consists of information of the quotient graph. However, doing so explicitly at each step of an IR algorithm

seems prohibitively expensive. Instead, we collect *some* information of the quotient graph during the execution of color refinement itself.

In particular, we collect precisely the information as described by the `ReportSplit` function (except for the entire coloring π). A standard technique [76] is to accumulate a hash of this information, and then use this hash as a node invariant. Traces introduced the notion of the *trace* invariant: we simply collect this information in a list, and use the entire list as a node invariant. In subsequent branches, instead of just writing a new trace and then comparing the results at the end of the computation, we can just compare the information on-the-fly each step of the way. (In particular, note that in subsequent branches, we do not even need to write a new list: we can simply track the position of where we are in the previously collected trace.) This enables an early-out of the color refinement algorithm, whenever the information differs.

4.4.3 Reversible Refinement

Sometimes, in an IR algorithm, we want to do color refinement, but then also be able to "undo" the application of color refinement again, efficiently. This corresponds to moving up and down in the IR tree. In order to facilitate this, we describe how a color refinement can be undone.

A trivial way to achieve this is to store the coloring π before applying color refinement, and then retrieving it if the refinement is to be undone. While this method seems unsophisticated, it turns out to be quite efficient in some cases. In particular, if color refinement changes most (i.e., a linear fraction) of the coloring anyway, then there really is no point in trying to be more granular. An obvious downside is that if color refinement does not change most of the coloring, we perform a linear time operation, even already *before* performing color refinement.

A more granular approach is to record the splits that are made while color refinement is performed. When we want to undo color refinement, we read the splits back and merge the colors that appear in the splits.

Recording the splits is simple enough: by implementing a `ReportSplit` function, we can maintain a list S of all the pairs (x, x') where x was split into x and x'. The algorithm to reverse the refinement is described in Algorithm 19. It is crucial to observe that at the time $\pi^{-1}(x')$ is accessed, the part of the coloring that concerns x' is correctly preserved. Note that the correctness of the algorithm also fundamentally relies on spacious colorings (see Section 2.2.4).

It is easy to see that Algorithm 19 runs in at most the time that it took the corresponding application of the color refinement algorithm. However, in practice, it is usually much faster since it does not need to consider the underlying graph or any auxiliary data structure (except S) at all.

4.4.4 Canonical and Non-Canonical Refinement

We can ease our requirements for color refinement, by only requiring the algorithm to produce an equitable coloring *up to renaming of colors*. Essentially, this still determines

Algorithm 19: Reverse a color refinement.

```
1 function ReverseRefinement
      Input: ➢ coloring π
             ➢ list of splits S
     Output: ≺ coarsens π according to S
2    // read S backwards
3    for ( each (x, x') of S in reverse order )
4        for ( each v in π^{-1}(x') )
5            VertexToCol_π[v] = x;
6            ColToSize_π[x'] -= 1;
7            ColToSize_π[x] += 1;
```

the same partitioning of vertices, but not the same *ordered* partitioning. In the context of IR algorithms this is generally not sufficient: it breaks the isomorphism-invariance of the IR tree. However, it is indeed sufficient as the *first* color refinement applied by the algorithm, before any individualizations are made.

The optimization is quite simple: we remove all sorting operations, i.e., Line 13 and Line 22 in Algorithm 10.

4.4.5 Matched Vertex Colorings

Matched vertex colorings are a technique introduced first by SAUCY [60]. The technique concerns the IR tree itself, however, the necessary computations needed for the technique are performed during color refinement itself. Since DEJAVU also makes use of the technique, we give a high-level description following the lines of [60].

Consider two vertex colorings π_1, π_2 of a graph G. We call π_1 and π_2 *isomorphic*, if for each $i \in \{1, \dots, n-1\}$ it holds that $|\pi_1^{-1}(i)| = |\pi_2^{-1}(i)|$ holds. This means π_1 and π_2 have the same set of non-trivial colors, and each corresponding color class has the same size. We call π_1 and π_2 *matched*, whenever $|\pi_1^{-1}(i)| \geq 2$ implies that $\pi_1^{-1}(i) = \pi_2^{-1}(i)$: all non-singleton color classes are identical.

Given two matched vertex colorings π_1, π_2, we may construct the following permutation $\varphi_{\pi_1,\pi_2} : V(G) \to V(G)$:

$$\varphi_{\pi_1,\pi_2}(v) := \begin{cases} v \text{ if } |\pi_1^{-1}(\pi_1(v))| \geq 2 \\ v' \text{ if } |\pi_1^{-1}(\pi_1(v))| = 1 \text{ and } \pi_1(v) = \pi_2(v') \end{cases}$$

Observe that when π_1, π_2 are matched, the above definition indeed defines a bijection. Vertices which are singletons are mapped onto each other, whereas vertices which occur in non-singleton color classes are mapped as the identity. (Observe that if v is a singleton in π_1, and $\pi_1(v) = \pi_2(v')$ holds, then v' must also be a singleton in π_1.)

We alter an observation of [60], to obtain the following lemma.

Lemma 67. *Let (G, π) be a vertex colored graph. Let ν_1, ν_2 be two nodes of $T_{\mathrm{CRef}}(G, \pi)$. Let π_1, π_2 be the corresponding equitable vertex colorings of ν_1, ν_2, with $\pi_1 \preceq \pi$ and $\pi_2 \preceq \pi$.*

If π_1 and π_2 are matched, then either φ_{π_1,π_2} is an automorphism of (G,π), or there is no automorphism of (G,π) which maps ν_1 to ν_2.

Proof. Note that by definition, $\varphi := \varphi_{\pi_1,\pi_2}$ respects the colors of the graph. In fact, any automorphism mapping ν_1 and ν_2 must map the singletons in the manner in which they are mapped in φ_{π_1,π_2}, otherwise the singleton color classes would not be respected. Assume that φ is no automorphism of (G,π). This means there must be an edge (v_1,v_2) such that $(v_1,v_2) \in E(G)$, whereas $(v_1,v_2)^\varphi$ is not. Without loss of generality, assume $v_1 \neq v_1^\varphi$.

If $v_2 = v_2^\varphi$, we observe that v_2 connects to v_1 of color c in (G,π_1), but v_2 does *not* connect to v_1^φ of color c in (G,π_2). Hence, we conclude that $v_2 \neq v_2^\varphi$ must hold for *any* automorphism mapping ν_1 to ν_2. Furthermore, we observe that since π_1 and π_2 are equitable, v_2 and v_2^φ can not obtain the same color in π_1 and π_2: the singleton v_1 connects to v_2 but not v_2^φ.

We conclude that $v_1, v_1^\varphi, v_2, v_2^\varphi$ are singleton. However, in this case, in order to respect the vertex coloring, these vertices must be mapped according to φ_{π_1,π_2}. Hence, if the edge relation of the graph is not satisfied by φ_{π_1,π_2}, then there is no automorphism mapping ν_1 to ν_2. □

Let τ be a target leaf of the IR tree. In subsequent color refinement computations of different branches, we can efficiently track whether the current vertex coloring is *matched* to the one of τ. Indeed, we can again implement this operation as a rather involved `ReportSplit` function. The details of the implementation are quite technical, but essentially, on each split, we check the vertices of the involved color classes. In turn, we remove and add differences between the color classes. (We refer to the implementation for more detail.)

A crucial fact however is that once we find a matched pair of vertex colorings, we can efficiently construct and check an automorphism. Let us first observe that it is easy to efficiently keep track of a list of singleton color classes using `ReportSplit`: if c or s is a new singleton of π, we add it to the list of singletons. When colorings are matched, we use the list of singletons to write φ_{π_1,π_2} into a dense-sparse permutation data structure (see Section 2.2.3). In turn, we verify φ_{π_1,π_2} on the graph using Algorithm 2. Given a matched vertex coloring and corresponding lists of singletons, we can therefore construct and test an automorphism in time

$$\mathcal{O}\left(\sum_{v \in \mathrm{supp}(\varphi_{\pi_1,\pi_2})} |N[v]|\right).$$

4.4.6 Small Graphs

We should mention that the data structures and algorithms used in this chapter are given with asymptotic scaling in mind. For small graphs, different considerations should be made. In particular NAUTY is quite efficient at dealing with these graphs, as was discussed in Section 2.4.1. For example, NAUTY features the use of adjacency matrices,

and the colorings are stored in a more rudimentary fashion. The colorings do not feature the VertexToLab or VertexToCol arrays, which has crucial implications for the color refinement implementation.

Chapter 5

Preprocessing

Many of the graphs and symmetries encountered in practical applications are of a peculiar type: on one hand, most of these graphs are quite simple in terms of their IR tree, i.e., there is only one equivalence class of leaves. On the other hand, the graphs of interest are often very large and may exhibit a lot of symmetry. At first, this might not seem like an issue: simple structure is, of course, handled "somewhat efficiently" by any well-implemented IR algorithm. In particular, if there is only one equivalence class of leaves, any IR algorithm should run in polynomial time. Considering the sheer scale of these practical graphs however, the approach starts to run into problems. The challenge arises to handle large, practical graphs more efficiently. Precisely this challenge prompted the design of SAUCY, which features special techniques precisely for these classes of graphs (in particular we mean matched vertex colorings, as described in Section 4.4.5). However, as was thoroughly discussed in Chapter 1, only having solvers available that are geared towards specific types of graphs is an undesirable situation.

Preprocessors are a commonly used paradigm to make solvers for computational problems more widely applicable. The idea is to shrink or simplify an input, before handing it to a main solver. The use of preprocessors has indeed already led to countless success stories, in particular in SAT, QBF or MaxSAT [36, 17, 65]. In these applications, it is standard to apply a preprocessor to all inputs. To the contrary, no preprocessor has been available for symmetry detection. In fact, McKay and Piperno [76] explicitly highlight that in their opinion "graphs of [particular types] ought to be handled by preprocessing" before using their tools. 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 Section 2.4.4). 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.

Preprocessor for Sparse Structure. We describe a preprocessor for symmetry detection that satisfies the specifications above. It is compatible by design with all state-of-the-art symmetry detection tools.

There are two immediate benefits to this approach: first of all, tackling certain parts of the graph independent of the rest of the solver severely reduces the software engineering burden, i.e., there is a separation of concerns. We can implement a fairly independent routine that deals with specific aspects of the graph. The second major benefit is that if designed correctly, the preprocessor should be compatible with *all* symmetry detection tools. We only need to implement these strategies once, and can then apply the benefits to all the other solvers as well.

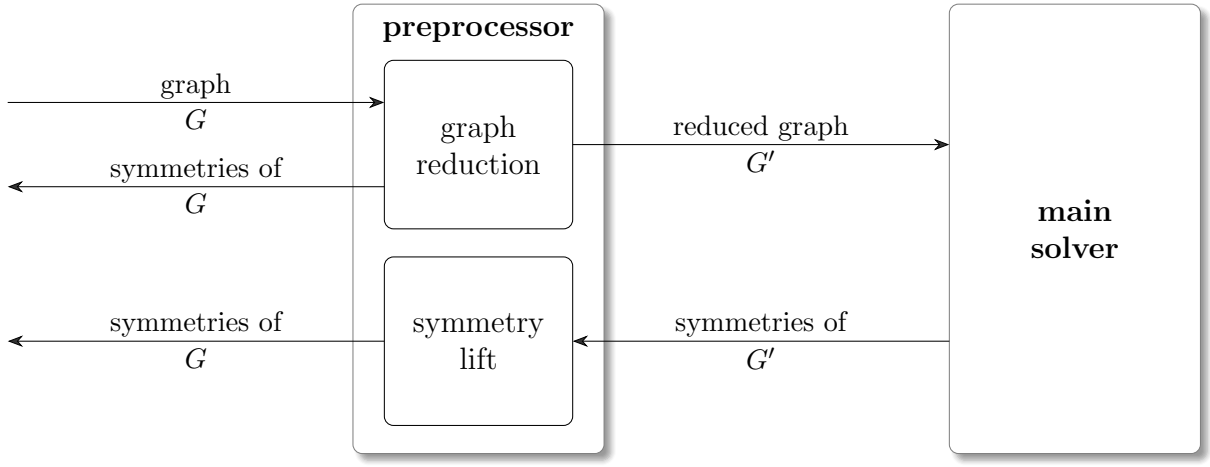


Figure 5.1: Our proposed preprocessor/main solver and user/preprocessor interfaces for symmetry detection. The preprocessor may already determine some (or all) symmetries of G during graph reduction. The reduced instance is then passed on to the main solver.

In this chapter, we describe the design and implementation of our preprocessor for sparse and large graphs. The preprocessor is an integral part of the DEJAVU solver and is applied to all inputs. However, the preprocessor also works in conjunction with any other state-of-the-art solver. In Section 7.3, we test the performance of our preprocessor in conjunction with all other state-of-the-art symmetry detection tools. In particular, we compare each solver without the preprocessor to itself with the preprocessor.

5.1 Interface and Conceptual Principles

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 is the preprocessor/main solver interface, as well as the user/preprocessor interface. 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 runtime of the main algorithm.

The design of our preprocessor is centered around the color refinement algorithm. Recall that color refinement is continuously and repeatedly applied in all state-of-the-art solvers (see Section 2.3). 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 symmetry detection 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 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 5.1.

We should remark that our implementation can also recover the size of the original automorphism group, given the size of the automorphism group of the reduced graph.

5.2 Framework for Reductions

Most, but not all, techniques we describe modify an input graph (G, π) on vertex set V to another graph (G', π') with vertex set $V' \subseteq V$ so that

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

Here, by $\mathrm{Aut}(G, \pi)|_{V'}$ we mean the set of maps obtained by restricting the domain of each $\varphi \in \mathrm{Aut}(G, \pi)$ to V' (and the range to $\varphi(V')$). If conditions (1) and (2) hold, V' must also be invariant under $\mathrm{Aut}(G, \pi)$. In conjunction, these conditions ensure that we do not introduce extra symmetry in the reduced graph, and the reduced graph's symmetries correspond to symmetries of the original graph.

Let us address a technicality: our data structures in Chapter 2 require us to store a graph using the vertex set $\{0, \ldots, n-1\}$, but clearly, the reduced graph G', as defined above, may not adhere to this. In the implementation, each reduction of the graph renames the vertices such that they map into $\{0, \ldots, n-1\}$. For all subsequently produced symmetries we apply a simple renaming step, to restore their original names. For the sake of simplifying our exposition here, we will not cover the renaming of vertices and let V' be any subset of V. Similarly, restricting the coloring π to $\pi|_{V'}$ leads to analogous issues, i.e., $\pi_{V'}$ is not spacious and colors are $\{0, \ldots, n-1\}$. Again, the implementation reorders vertex colorings appropriately, while in our exposition here, we will not.

Under the conditions stated above the restriction to V' is a natural homomorphism

$$p\colon \mathrm{Aut}(G, \pi) \to \mathrm{Aut}(G', \pi').$$

The orbit-stabilizer theorem (Theorem 7) implies then that if $S' \subseteq \mathrm{Aut}(G, \pi)$ is a set of lifts of a generating set S of $\mathrm{Aut}(G', \pi')$, i.e. $p(S') = S$, then

$$\mathrm{Aut}(G, \pi) = \langle S', \ker(p) \rangle.$$

Here

$$\ker(p) = \{\varphi \in \mathrm{Aut}(G, \pi) \mid p(\varphi) = \mathrm{id}\}$$

is the *kernel* of p.

Overall this enables us to separate the computation of $\mathrm{Aut}(G, \pi)$ into computing automorphisms of the removed parts of the graph and the automorphisms of the reduced graph. To summarize, crucial for these techniques are the two following ingredients:

1. The reduced graph (G', π') and a generating set for the reduced parts $\ker(p)$ can be efficiently computed from (G, π) (the *reduction* step).
2. The set of lifts S' can be efficiently computed from a given generating set $\langle S \rangle = \mathrm{Aut}(G', \pi')$ of the automorphism group of the reduced graph (the *lifting* step).

5.3 Automorphism-Preserving Reductions

Let us first discuss reductions which preserve all automorphisms, i.e., the cases in which the lift is trivial.

5.3.1 Color Refinement and Discrete Vertices

The first step in our procedure is to apply color refinement, and in turn reduce discrete vertices from the graph. It should be noted that this step is a prerequisite for most of the following routines: subsequent routines often rely on the fact that the given vertex coloring is equitable.

Color Refinement. Recoloring a graph (G, π) with the coarsest equitable coloring preserves all the symmetries of the graph (see Section 2.3). Thus, this a correct way of preprocessing the graph. In fact, we may even use non-canonical color refinement (see Section 4.4.4).

Singleton Vertices. Recall that singleton vertices are those vertices v of (G, π) with a color class size of 1, i.e., $|\pi^{-1}(\pi(v))| = 1$. Since automorphisms must be color-preserving, for all automorphisms $\varphi \in \mathrm{Aut}(G, \pi)$ and all singletons v of (G, π) it holds that $\varphi(v) = v$. Indeed, it is easy to see that we may just remove them from the graph entirely, and this operation preserves the automorphism group of the graph.

Let V_d denote the set of singleton vertices of (G, π). Consequently, we reduce G to induced subgraph $G' \coloneqq G[V(G) \setminus V_d]$, and π to the restriction $\pi' \coloneqq \pi|_{V(G) \setminus V_d}$. It is easy to see that $\mathrm{Aut}(G, \pi)|_{V'} = \mathrm{Aut}(G', \pi')$ holds, and $\mathrm{Aut}(G', \pi')$ lifted to the identity on V_d is $\mathrm{Aut}(G, \pi)$.

5.3.2 Quotient Graph Flips

Let us assume (G, π) is our graph of consideration. We describe how to *flip edges* between color classes. Let C_1, C_2 be two distinct color classes of π. Essentially, if π is equitable, these are two nodes of $Q(G, \pi)$. 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. Formally, we transform (G, π) into (G', π) for an edge flip between color classes C_1 and C_2 of π. We let $V(G') \coloneqq V(G)$. For the edges, we define

$$\begin{aligned} E(G') \coloneqq &\{(v_1, v_2) \mid (v_1, v_2) \in E(G), \neg(v_1 \in C_1 \wedge v_2 \in C_2) \wedge \neg(v_1 \in C_2 \wedge v_2 \in C_1)\} \\ &\cup \{(v_1, v_2) \mid v_1 \in C_1, v_2 \in C_2, v_1 \neq v_2, (v_1, v_2) \notin E(G)\} \\ &\cup \{(v_2, v_1) \mid v_1 \in C_1, v_2 \in C_2, v_1 \neq v_2, (v_1, v_2) \notin E(G)\} \end{aligned}$$

Since this operation is isomorphism-invariant and reversible, the automorphism group of the graph does not change. We record this fact in the following lemma.

Lemma 68. *Let (G, π) be a vertex colored graph, and C_1 and C_2 two (not necessarily distinct) color class of (G, π). Let (G', π) denote the graph obtained having performed the edge flip between C_1 and C_2. It then holds that* $\mathrm{Aut}(G, \pi) = \mathrm{Aut}(G', \pi)$.

Proof. Let $\varphi \in \mathrm{Aut}(G, \pi)$. Assume towards a contradiction $\varphi \notin \mathrm{Aut}(G', \pi)$. Since the vertex colorings of the considered graphs are identical, φ is color-preserving. Hence, there is an edge $(v_1, v_2) \in E(G')$ such that $(v_1, v_2)^\varphi \notin E(G')$. Since we only manipulated edges between vertices of C_1, C_2, we can assume without loss of generality $v_1 \in C_1$ and $v_2 \in C_2$. Due to our construction, we can conclude $(v_1, v_2) \notin E(G)$ and $(v_1, v_2)^\varphi \in E(G)$. Hence, φ does not respect the edge relation of G, a contradiction to the assumption $\varphi \in \mathrm{Aut}(G, \pi)$. The other direction follows analogously. □

5.4 Lifts based on Vertices

Let us now turn to graph reductions which may require a non-trivial lift. During preprocessing, some parts removed 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 the reduced graph's symmetries back to symmetries of the original graph. In particular, the lifted symmetries must map all the removed parts correctly. These reductions are facilitated by a general technique to lift automorphisms, called the canonical representation strings. We begin by introducing canonical representation strings, followed by a description of how they can be used to lift automorphisms.

Canonical Representation Strings. 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* $\mathcal{R} : V \mapsto V^*$ from the vertices to sequences of vertices as follows. Assume we have a graph $(G, \pi) \coloneqq ((V, E), \pi)$ which is reduced to $(G', \pi) \coloneqq ((V', E'), \pi')$ with $V' \subseteq V$ and $E' \subseteq E$. We require the following:

1. It holds that $\mathcal{R}(v) \coloneqq vS$ with $S \in V^*$ for all $v \in V'$, i.e., each remaining vertex must represent itself first.

2. It holds that $\mathcal{R}(v) \coloneqq \epsilon$ 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 $v \coloneqq \mathcal{R}(v')_i$, i.e., each deleted vertex is represented by at most one remaining vertex, once.

Automorphism Group Lift. We explain how canonical representation strings can be used to lift automorphisms from the reduced graph to the original graph. We use our dense-sparse encoding of automorphisms, as was introduced in Section 2.2.3. A global dense-sparse encoding is maintained as the identity, and algorithms write automorphisms to this data structure whenever necessary. In turn, this automorphism is either utilized or presented as an output to the user and lastly reset back to the identity.

For each automorphism of the remaining graph $\varphi \in \mathrm{Aut}(G')$, we now define its *lifted bijection* $\varphi_{\mathcal{R}} \in \mathrm{Sym}(V)$. First, we require that

$$|\mathcal{R}(v)| = |\mathcal{R}(v^{\varphi})|$$

holds for every $v \in V'$, otherwise we can not construct a lifted bijection. We define

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

Using a canonical representation mapping $\mathcal{R}$ and a dense-sparse automorphism encoding, 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. This procedure is described in more detail in Algorithm 20.

Description of Algorithm 20. The algorithm reads as input a permutation φ and canonical representation strings $\mathcal{R}$. An auxiliary dense-sparse permutation data structure φ' is assumed to be set to the identity. The algorithm, in turn, constructs $\varphi_{\mathcal{R}}$ in the data structure φ'. Then, $\varphi_{\mathcal{R}}$ is presented as an "output" to the user or further routines, and finally reset to the identity.

Algorithm 20: Lift permutation according to canonical representation strings.

```
1  function LiftAutomorphisms
       Input: ➢ canonical representation strings R
              ➢ dense-sparse permutation φ
      Output: ≺ output φ_R using φ'
   Auxiliary: ✏ dense-sparse permutation φ', where φ' = id
2    // using dense-sparse permutation, we can iterate the support
3    for ( each vertex v in supp(φ) )
4      // map repr. string of v to repr. string of v^φ
5      foreach i ∈ {0, ..., |R(v)| − 1} do
6        // map vertices accordingly...
7        Perm_φ'(R(v)[i]) := R(φ(v))[i];
8        // ...but also maintain the support
9        if R(v) ≠ R(φ(v)) then append R(v)[i] to Supp_φ' ;
10   present φ' as output to the user;
11   reset φ';
```

Runtime of Algorithm 20. By a simple inspection of Algorithm 20, it follows that given $\varphi \in \mathrm{Aut}(G')$, the algorithm runs in worst-case time $\mathcal{O}(|\operatorname{supp}(\varphi_{\mathcal{R}})|)$.

We note that 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.)

We want to remark that in the implementation, we use *one* canonical representation mapping to keep track of all removed vertices. 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 $\{0, 1, \ldots, n-1\}$, whenever n vertices remain.

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

In the following sections, we describe reduction rules based on canonical representation strings.

5.4.1 Degree 0

Preprocessing vertices of degree 0 (and analogously $n-1$) is simple. The algorithm detects and treats color classes solely consisting of vertices of degree 0, one class at a

Algorithm 21: Preprocess degree 0 vertices of a given graph.

```
1 function PreprocessDeg0
      Input: ➢ graph G
             ➢ equitable coloring π of G
     Output: ≺ removes degree 0 vertices from G
             ≺ generators for all the removed degree 0 vertices
  Auxiliary: ✏ dense-sparse permutation φ', where φ' = id
2   for ( each color c in π(V(G)) )
3       pick arbitrary vertex v of color c;
4       if deg(v) = 0 then
5           for ( each vertex v' in π^{-1}(c) with v ≠ v' )
6               output generator (vv') using φ';
7               reset φ';
8           remove vertices π^{-1}(c) from G;
```

time. 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 procedure is described in Algorithm 21.

Description of Algorithm 21. The kernel $\ker(p)$ of the restriction p onto V' is computed as follows. For each color class of degree 0 vertices in (G, π) we output generators for the symmetric group on the class.

Runtime of Algorithm 21. The algorithm can be implemented to run in time $\mathcal{O}(|\pi| + n_0)$, where n_0 refers to the number of vertices of degree 0 in the graph.

We note that loop of Line 5 describes how to output the generators for a natural symmetric action on a set of vertices. In the following, we just refer to this operation as producing the generators for a natural symmetric action.

5.4.2 Twins

Two vertices $v, v' \in V(G)$ are called *true twins*, whenever they have the same neighborhood in G, i.e., $N(v) = N(v')$ holds. Similarly, two vertices $v, v' \in V(G)$ are called *false twins*, whenever they have the same closed neighborhood in G, i.e., $N[v] = N[v']$ holds. For a vertex colored graph (G, π), let us further require that twins must be of equal color, i.e., $\pi(v) = \pi(v')$ must hold in addition to the above definition. For both true and false twins, it follows immediately that there is an automorphism $\varphi \in \mathrm{Aut}(G, \pi)$ that interchanges v and v'. Specifically, the automorphism may map $\varphi(v) = v'$ and $\varphi(v') = v$, and be the identity everywhere else.

Using a folklore partition-refinement approach, the partition of all true or false twins of a graph can be determined in time $\mathcal{O}(n + m)$: we initialize a partition-refinement structure to the coloring π, and refine it with respect to the open (closed) neighborhood

of every vertex in G. The partitioning in turn corresponds to equivalence classes of true (false) twins. (This process can be terminated early, whenever the partitioning becomes discrete.) For our graph reduction, we then remove all but one of the vertices in each equivalence class. We output generators for a natural symmetric action on each class of twins. The remaining vertex is colored with a color indicating its original color as well as the size of its twin class.

Let us describe and argue correctness of the procedure formally. We do so for *true* twins, but the arguments follow analogously for false twins. Observe that being true twins defines an equivalence relation t on the vertices of a graph. We refer to $T = [v]_t$ for some $v \in V(G)$ as an *equivalence class* of true twins. In turn, $V(G)$ can be partitioned into equivalence classes of true twins. For every equivalence class of true twins, we define an (arbitrary) canonical representative $r(T)$.

Given a graph (G, π), let us define the reduced graph (G', π'). We define $V' = \{r([v]_t) \mid v \in V(G)\}$. The reduced graph G' is then the induced subgraph $G' = G[V']$. To ease notation, our coloring π' will map vertices to tuples: a tuple consists of a color of π, the original color of the vertex, as well as the size of their equivalence class. Formally, $\pi'(v) \coloneqq (\pi(v), |[v]_t|)$. Every remaining vertex represents its equivalence class of twins, i.e., for $v \in V'$ we let

$$\mathcal{R}(v) \coloneqq v([v]_t \setminus \{v\}).$$

We observe that the reduction is indeed *not* symmetry-preserving, due to the fact that symmetries of the equivalence classes of twins themselves, in particular those which map the canonical representative, are not preserved. However, we argue correctness of the reduction below.

Let $\langle S \rangle = \mathrm{Aut}(G', \pi')$ and S' be the respective generating set lifted using $\mathcal{R}$. We argue that the original automorphism group $\mathrm{Aut}(G, \pi)$ is generated by S', and $\mathrm{Sym}(T)$ for each equivalence class of twins T.

Lemma 70. *Let (G', π') be the graph obtained by the reduction of true twins from (G, π), and $\mathcal{R}$ the canonical representation strings. Let $\langle S \rangle = \mathrm{Aut}(G', \pi')$, and $S' \coloneqq \{\varphi_{\mathcal{R}} \mid \varphi \in S\}$ be the respective generating set lifted using $\mathcal{R}$. It then holds that*

$$\mathrm{Aut}(G, \pi) = \langle S' \cup \{\mathrm{Sym}([v]_t) \mid v \in V(G)\} \rangle.$$

Proof. Observe that if $v^\varphi = v'$ holds, then $|[v]_t| = |[v']_t|$ follows. Hence, the coloring of (G', π') does not restrict any automorphism of (G, π).

Assume $\varphi \in \langle S' \cup \{\mathrm{Sym}([v]_t) \mid v \in V(G)\} \rangle$. First, assume $\varphi \in \mathrm{Sym}([v]_t)$ for some $v \in V(G)$. Clearly, since all vertices of $[v]_t$ are twins in (G, π), $\varphi \in \mathrm{Aut}(G, \pi)$ follows. Let us now consider the case $\varphi \in S'$. Let $(v_1, v_2) \in E(G)$. Clearly both $[v_1]_t \neq [v_1^\varphi]_t$ and $[v_2]_t \neq [v_2^\varphi]_t$ must hold, since the statement follows trivially for true twins. Furthermore, we know $|[v_1]_t| = |[v_1^\varphi]_t|$ and $|[v_2]_t| = |[v_2^\varphi]_t|$ hold. Consider the edge $(r([v_1]_t), r([v_2]_t)) \in E(G')$. By construction of the lift using $\mathcal{R}$, it follows that $(r([v_1]_t), r([v_2]_t))^\varphi \in E(G')$ holds. We recall that v_1 and $r([v_1]_t)$, as well as v_2 and $r([v_2]_t)$ are true twins. Therefore, $(v_1, v_2)^\varphi \in E(G)$ holds.

For the other direction, assume $\varphi \in \mathrm{Aut}(G, \pi)$. We observe that the equivalence classes of twins must be (set-wise) preserved by φ. All automorphisms mapping these equivalence

classes onto each other can be recovered using S'. Any desired ordering of the equivalence classes themselves can be recovered using $\{\mathrm{Sym}([v]_t) \mid v \in V(G)\}$. □

5.4.3 Degree 1

Exhaustively removing all vertices of degree 1 essentially removes all tree-like appendages from graphs. Fast algorithms for tree isomorphism have been known for decades [1]. Moreover, 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 (see [63]).

We remove degree 1 vertices recursively. Let (G, π) be a graph that contains degree 1 vertices, where π is required to be equitable. We describe (G', π') 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 $\mathcal{R}(p_i) \coloneqq p_i C_i$ (where C_i may appear in arbitrary order). We set $G' \coloneqq G[V(G) \setminus \{C\}]$. Again, the coloring π is simply reduced to the domain of G', i.e., $\pi' \coloneqq \pi|_{V(G')}$.

We collect the following properties of the reduction.

Lemma 71. *Let (G', π') be the graph obtained by the reduction of a color class of degree 1 vertices C from (G, π). Let $\mathcal{R}$ denote the respective canonical representation strings. The following hold true.*

1. *The coloring π' is equitable on G'.*
2. *The reduction is symmetry-preserving and symmetry-lifting, i.e.,*

$$\mathrm{Aut}(G, \pi)|_{V'} = \mathrm{Aut}(G', \pi')$$

   holds.

3. *Let p denote the natural homomorphism which corresponds to the reduction. Let $C_1, \dots, C_m$ denote the partition of C, where $c \in C_i$ is adjacent to $p_i \in P$. 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).*
4. *Let $\langle S \rangle = \mathrm{Aut}(G', \pi')$, then $S' \coloneqq \{\varphi_{\mathcal{R}} \mid \varphi \in S\}$ satisfies $S' \subseteq \mathrm{Aut}(G, \pi)$ and $p(S') = S$.*

Proof. For claim (1), observe that we only remove the color class C from G, all neighbors of which are in color class P. Since we remove *all* vertices of the color class C, no further split of P becomes possible. Hence, π' is equitable.

For claim (2), $\mathrm{Aut}(G,\pi)|_{V'} \subseteq \mathrm{Aut}(G',\pi')$ immediately follows from the fact that we simply remove an entire color class. For the other direction, assume $\varphi \in \mathrm{Aut}(G',\pi')$. Since π is equitable, observe that the color of each $p \in P$ encodes the number of children in C that were removed, i.e., each $p \in P$ must have the same number of neighbors (children) in C In particular, for each $p \in P$, we removed the same number of degree 1 children of color C. Hence, we can simply lift φ to an automorphism of $\mathrm{Aut}(G,\pi)$ by mapping the children of each p and p' accordingly. Note that this is formally described by $\varphi_{\mathcal{R}}$.

Claim (3) follows by a simple inspection: if an automorphism $\varphi \in \mathrm{Aut}(G,\pi)$ maps $v, v' \in C$ as $\varphi(v) = v'$, but $N(v) = \{p\}$ and $N(v') = \{p'\}$ with $p \neq p'$ holds, then $\varphi(p) = p'$ follows. Hence, $\varphi \notin \ker(p)$.

For claim (4), for $\varphi \in \mathrm{Aut}(G',\pi')$, as discussed above, we observe that $\varphi_{\mathcal{R}}$ simply lifts mappings of the parents to their respective children. Hence, $\varphi_{\mathcal{R}} \in \mathrm{Aut}(G,\pi)$ follows immediately. Moreover, since $\varphi_{\mathcal{R}}|_{V(G')} = \varphi$ holds, the claim follows. □

The process can then be repeated until all vertices of degree 1 are removed. (Observe that iteration is enabled by the first claim of Lemma 71.) A sketch of the iterated procedure can be found in Algorithm 22. Correctness of the algorithm follows from an iterated application of the above reduction and Lemma 71.

5.4.4 Degree 2 with Unique Endpoints

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 [62, Proof of Lemma 15]).

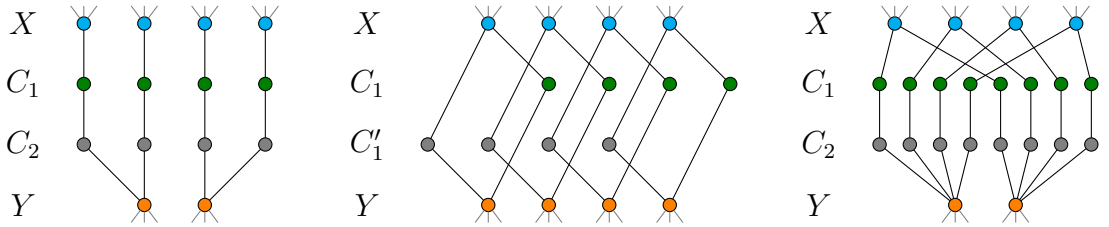
There are, however, drawbacks to this approach: most solvers do not implement directed and colored edges. Since we want our preprocessor to be compatible with all modern solvers, this immediately disallows the use of directed, colored edges. Even when they do, using directed and colored edges comes at the price of additional overhead [94]. 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. We describe particular settings in which this is possible.

Non-branching paths with unique endpoints. 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

Algorithm 22: Preprocess degree 1 vertices of a given graph.

```
1  function PreprocessDeg1
       Input: ➢ graph G
              ➢ equitable coloring π of G
      Output: ≺ removes degree 1 vertices from G
              ≺ generators for kernel of the reduction
   Auxiliary: ✏ dense-sparse permutation φ', where φ' = id
2      // maintain a worklist on S
3      initialize empty stack S;
4      // find the colors with degree 1
5      for ( each color c in π(V(G)) )
6          pick arbitrary vertex v of color c;
7          if deg(v) = 1 then
8              push c to S;
9      // continue as long as the worklist is not empty...
10     while stack S non-empty do
11         pop c from S;
12         // vertices of c are essentially the leaves of a tree
13         pick arbitrary vertex v of color c;
14         if deg(v) = 1 and π(v) ≠ π(v') where v ~ v' then
15             // which of these leaves have the same parent?
16             make a partition ξ of π^{-1}(c) such that ξ(v) = ξ(v') iff N(v) = N(v');
17             // for leaves with same parent, output symmetric action
18             for ( each part x of ξ )
19                 write generator for symmetric action on ξ^{-1}(x) to φ';
20                 output lift φ'_R;
21                 reset φ';
22             for ( each part x of ξ )
23                 // all vertices of ξ^{-1}(x) have the same parent
24                 let p be parent of ξ^{-1}(x);
25                 // add the leaves to representation string of their
                      parent
26                 for ( each v in π^{-1}(x) )
27                     append R(v) to R(p);
28                     R(v) := ε;
29             // now remove all these leaves
30             remove π^{-1}(c) from G;
31             // the parents might have degree 1 now
32             put parent color c' on stack S;
```



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

Figure 5.2: Reducible degree 2 patterns.

from the set of edges. In particular, the inserted edges may not interfere with existing edges.

Let (G, π) denote our graph of interest. We consider color classes of π. 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 5.2a). More formally, suppose $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ are color classes of π 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},
4. and every node in X has exactly one neighbor in C_1.

If the above requirements are satisfied, we say that $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ is a set of paths with a unique endpoint in X. Then, we define the reduced graph $G' = (V', E')$ via $V' \coloneqq V \setminus (C_1 \cup \cdots \cup C_t)$ 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 $\mathcal{R}(x) = xc_1c_2 \cdots c_t$, 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.

Lemma 72. *Let (G', π') be the graph obtained when reducing the set of paths $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ with a unique endpoint in X from (G, π). Let $\mathcal{R}$ denote the respective canonical representation strings. The following hold true.*

1. *The reduction is symmetry-preserving and symmetry-lifting, i.e.,*

$$\mathrm{Aut}(G, \pi)|_{V'} = \mathrm{Aut}(G', \pi')$$

   holds.

2. *Let p denote a natural homomorphism which corresponds to the reduction. The kernel of p is trivial, i.e.,* $\ker(p) = \{\mathrm{id}\}$.

3. *Let* $\langle S \rangle = \mathrm{Aut}(G', \pi')$, *then* $S' \coloneqq \{\varphi_{\mathcal{R}} \mid \varphi \in S\}$ *satisfies* $S' \subseteq \mathrm{Aut}(G, \pi)$ *and* $p(S') = S$.

Proof. For claim (1), let us first assume $\varphi \in \mathrm{Aut}(G, \pi)|_{V'}$. Clearly, φ respects the edge relation of G' for all edges not connecting the vertices of X and Y. For the edges between X and Y, observe that each edge corresponds to a unique, removed path of G. Since φ respects these paths, it also respects the edge relation between X and Y. The other direction is analogous: since X and Y are not adjacent in G, the edges of G' only need to ensure that automorphisms respect the paths in G.

For claim (2), observe that an automorphism mapping any path p to another path p', must map the corresponding endpoints in X onto each other. Hence, there are no automorphisms in the kernel.

For claim (3), observe that $\varphi \in \mathrm{Aut}(G', \pi')$ determines how the paths have to be mapped by determining $\varphi(X)$. In turn, $\varphi_{\mathcal{R}}$ simply lifts the mapping of X to a mapping of the paths. Both claims follow immediately. □

Moreover, it is easy to check that if π is equitable, then π' is equitable after application of the reduction.

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 5.2b). 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 quite common, in particular on MIP and SAT benchmarks.

5.5 Lifts based on Edges

Edge Representation Strings. We introduce an extension of representation strings, which are associated with the remaining *edges*. We define this extension formally by an *edge representation mapping* $\mathcal{E} : V \times V \mapsto V^*$, which maps an edge of the remaining graph to a sequence of vertices of the original graph. Assume we have a graph $(G, \pi) \coloneqq ((V, E), \pi)$ which is reduced to $(G', \pi) \coloneqq ((V', E'), \pi')$ with $V' \subseteq V$ and $E' \subseteq E$. We require the following:

1. It holds that $\mathcal{E}(v_1, v_2) = \epsilon$ for all $v_1 \in V \setminus V'$ or $v_2 \in V \setminus V'$, i.e., a removed edge does not represent any vertex.

2. For each deleted vertex $v \in V \setminus V'$ there is at most one $(v_1, v_2) \in V' \times V'$ and at most one $i \in \mathbb{N}$ such that $v = \mathcal{E}(v_1, v_2)_i$, i.e., each deleted vertex is represented by at most one remaining (directed) edge, once.

Automorphism Group Lift. We extend our automorphism group lift to lift automorphisms from the reduced graph to the original graph. For each automorphism of the remaining graph $\varphi \in \mathrm{Aut}(G', \pi')$ we define its *lifted bijection* $\varphi_{\mathcal{E}} \in \mathrm{Sym}(V)$. Again, we require that

$$|\mathcal{E}(v_1, v_2)| = |\mathcal{E}(v_1^{\varphi}, v_2^{\varphi})|$$

holds for every $\varphi \in \mathrm{Aut}(G', \pi')$ and $(v_1, v_2) \in V' \times V'$, otherwise we can not construct a lifted bijection. We define

$$\varphi_{\mathcal{E}}(v) \coloneqq \begin{cases} \varphi(v) & \text{if } v \in V' \\ \mathcal{E}(v_1^{\varphi}, v_2^{\varphi})_i & \text{if } v = \mathcal{E}(v_1, v_2)_i \text{ for } (v_1, v_2) \in V' \times V', i \in \mathbb{N} \\ v & \text{if } v \neq \mathcal{E}(v_1, v_2)_i \text{ for all } (v_1, v_2) \in V' \times V', i \in \mathbb{N}. \end{cases}$$

Observe that technically, the "edges" used by $\mathcal{E}$ need not correspond to the actual remaining edges in the reduced graph.

In the implementation, we store edge representation maps as a sparse graph (see Section 2.2.2) on the original vertices of the graph. This enables the implementation to efficiently look up edge representation maps incident to vertices in the support of a given permutation.

5.5.1 Degree 2 and Edge Flips

Another case that can be handled efficiently and is not covered by previous techniques is where color classes X and Y ($X \neq Y$) 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 5.2c). It is easy to see that deleting all such paths is both symmetry preserving and symmetry lifting. Observe that this is strongly related to the edge flip described in Section 5.3.2.

Formally, suppose $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ are distinct 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. Given a graph (G, π), we define the reduced graph for a given potential edge flip $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$. We define the reduced graph $G' = G[V \setminus (C_1 \cup \cdots \cup C_t)]$, and π' is restricted accordingly. For each pair $x \in X$ and $y \in Y$, let $(x, c_1, \ldots, c_t, y)$ denote the unique path connecting x to y. The corresponding edge representation map is $\mathcal{E}(x, y) = c_1 c_2 \cdots c_t$. We argue correctness of the reduction.

Lemma 73. *Let (G', π') be the graph obtained by reducing the edge flip*

$$X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$$

in (G, π). Let $\mathcal{E}$ denote the respective edge representation strings. The following hold true.

1. *The reduction is symmetry-preserving and symmetry-lifting, i.e.,*

$$\mathrm{Aut}(G,\pi)|_{V'} = \mathrm{Aut}(G',\pi')$$

holds.

2. *Let p denote a natural homomorphism which corresponds to the reduction. The kernel of p is trivial, i.e.,* $\ker(p) = \{\mathrm{id}\}$.

3. *Let $\langle S\rangle = \mathrm{Aut}(G',\pi')$, then $S' \coloneqq \{\varphi_{\mathcal{E}} \mid \varphi \in S\}$ satisfies $S' \subseteq \mathrm{Aut}(G,\pi)$ and $p(S') = S$.*

Proof. For claim (1), $\mathrm{Aut}(G,\pi)|_{V'} \subseteq \mathrm{Aut}(G',\pi')$ follows immediately since we are simply removing a union of color classes from the graph. Observe that $C_1,\dots,C_t$ does not restrict any permutation of X and Y, and is analogous to a homogeneous connection of X and Y (compare with Lemma 68). Hence, all $\varphi \in \mathrm{Aut}(G',\pi')$ correspond to a symmetry of $\mathrm{Aut}(G,\pi)$, by lifting the mapping of the endpoints to the paths, as is described formally by $\varphi_{\mathcal{E}}$.

For claim (2), observe that an automorphism mapping any path p to another path p', must map the corresponding endpoints in X and Y onto each other. Since there are no two paths which have the same endpoint in X *and* Y, it follows that there are no automorphisms in the kernel.

For claim (3), observe that $\varphi \in \mathrm{Aut}(G',\pi')$ determines how the paths have to be mapped by determining $\varphi(X)$ and $\varphi(Y)$: assume a symmetry maps $x \in X$ to $x' \in X$ and $y \in Y$ to $y' \in Y$. This just means that in the lift, we need to map the path connecting x to y to the path connecting x' to y'. This matches the definition of $\varphi_{\mathcal{E}}$. Both claims follow immediately. □

Observe that indeed, canonical representation strings are not sufficient to express the lift: we need to determine how $C_0, C_1, \dots, C_t$ are mapped, and this depends on *both* the vertices of X and Y.

5.5.2 Degree 2 Densification

As a last resort, we use a technique which "summarizes" the out-going paths of a particular color from a vertex into an additional hub vertex. Figure 5.3 illustrates the technique. The technique works without many of the requirements necessary for the previous techniques. However, while it is still able to substantially reduce the size of graphs, compared to the other techniques, it is the least efficient in doing so. Another technicality is that it introduces new vertices which means we need to alter our framework slightly in order to argue correctness. Let us first describe the technique in more detail. Formally, suppose $X = C_0, C_1, \dots, C_t, C_{t+1} = Y$ are distinct colors so that

1. for $i \in \{1,\dots,t\}$ vertices in C_i have degree 2,

2. for $i \in \{1,\dots,t\}$ vertices in C_i have a neighbor in C_{i-1} and C_{i+1}, and

3. if $(x, c_1, \ldots, c_t, y)$ is a path where $c_i \in C_i$, then there is no distinct path

$$(x, c'_1, \ldots, c'_t, y) \neq (x, c_1, \ldots, c_t, y)$$

connecting endpoints x and y.

If these requirements are met, we call $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ a *set of distinct paths*.

Given a graph (G, π), we define the reduced graph (G', π') for a given set of distinct paths $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$. Let us first define the set of *hub vertices* $H \coloneqq \{h_x \mid x \in X\}$, and we require that the names of vertices H are fresh, i.e., $H \cap V(G) = \emptyset$. Hence, with each $x \in X$ we associate a new hub vertex $h_x \in H$. Let us now formally define the reduced graph. The vertex set is

$$V' \coloneqq (V \cup H) \setminus (C_1 \cup \cdots \cup C_t),$$

i.e., we remove paths but introduce the hub vertices H. On vertices $V' \cap V$, π' is restricted accordingly, and hub vertices H receive a fresh color not appearing in π. In other words, H is a color class in π'. For each pair $x \in X$ and $y \in Y$, let $(x, c_1, \ldots, c_t, y)$ denote the unique path connecting x to y, where $c_i \in C_i$. The edge set is

$$E' \coloneqq E(G[V' \setminus H]) \cup \{(h_x, x) \mid x \in X\} \cup \{(x, h_x) \mid x \in X\} \cup E''.$$

E'' includes (h_x, y) (and (y, h_x), respectively) for each $h_x \in H$ and $y \in Y$, if and only if there is a path $(x, c_1, \ldots, c_t, y)$ in G. The corresponding edge representation map is $\mathcal{E}(h_x, y) = c_1 c_2 \cdots c_t$.

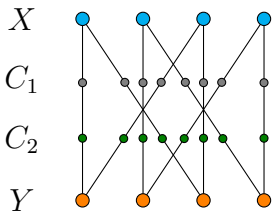
For the lift, we remove the vertices of H from the symmetry but otherwise simply apply $\mathcal{E}$. We argue correctness of the reduction, by slightly altering our definitions of Section 5.2.

Lemma 74. *Let (G', π') be the graph obtained by reducing the set of distinct paths $X = C_0, C_1, \ldots, C_t, C_{t+1} = Y$ in (G, π). Let $\mathcal{E}$ denote the respective edge representation strings. The following hold true.*

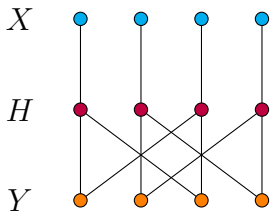
1. *It holds that*
$$\mathrm{Aut}(G, \pi)|_{V' \setminus H} = \mathrm{Aut}(G', \pi')|_{V' \setminus H}.$$

2. *The restriction from V to $V' \setminus H$ is a natural homomorphism*
$$p\colon \mathrm{Aut}(G, \pi) \to \mathrm{Aut}(G', \pi')|_{V' \setminus H}.$$
The kernel of p is trivial, i.e., $\ker(p) = \{\mathrm{id}\}$.

3. *Let $\langle S \rangle = \mathrm{Aut}(G', \pi')$, then $S' \coloneqq \{\varphi_{\mathcal{E}} \mid \varphi \in S\}$ satisfies $S'|_V \subseteq \mathrm{Aut}(G, \pi)$ and $p(S'|_V) = S|_{V' \setminus H}$.*



(a) Colors without duplicate paths.



(b) Reduced using hub vertices.

Figure 5.3: Example for degree 2 "densification" strategy, which introduces a hub vertex to summarize the out-going paths of a particular vertex.

Proof. For claim (1), we simply observe that any automorphism of G must respect the edge relation of the set of distinct paths, whereas automorphisms of G' must respect the edge relation between color class X and H – a matching – and between Y and H, which encodes precisely the set of paths in G. The claim follows.

For claim (2), observe that an automorphism mapping any path p to another path p', must map the corresponding endpoints in X and Y onto each other. Since there are no two paths which have the same endpoint in X *and* Y, it follows that there are no automorphisms in the kernel.

For claim (3), observe that $\varphi \in \mathrm{Aut}(G', \pi')$ determines how the paths have to be mapped by determining $\varphi(H)$ and $\varphi(Y)$: assume a symmetry maps $h_x \in H$ to $h_{x'} \in H$ (which implies $x \in X$ is mapped to $x' \in X$) and $y \in Y$ to $y' \in Y$. This just means that in the lift, we need to map the path connecting x to y to the path connecting x' to y'. This matches the definition of $\varphi_{\mathcal{E}}$.

□

5.6 Further Techniques

We discuss a few further techniques, which are currently *not* implemented within the preprocessor.

5.6.1 Non-uniform Components

Consider the quotient graph $Q = Q(G, \pi)$ of a graph G with respect to an equitable 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 75. *If $D_1, \ldots, D_t$ denote the represented vertices of 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 [63]).

These types of components have previously been employed for isomorphism and automorphism testing [45, 59]. (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.

Using non-uniform components, we can therefore partition the initial symmetry detection problem into several independent problems. Therefore, the technique can be used to make several independent solver calls. Usually, only one component is left, or there is one very large component and several smaller ones. On some practical graphs however, the technique can be quite effective.

It should be noted that in the implementation, the routine is not implemented as part of the preprocessor, but rather as part of the DEJAVU algorithm (see Chapter 6).

5.6.2 Probing

The initial publication of the preprocessor featured a technique called *probing* [7]. Using IR, the probing strategy only searches for automorphisms that 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.) The probing strategy only attempted to detect automorphisms that can be determined "early" using IR (see Section 4.4.5).

The technique is particularly effective for NAUTY, BLISS, and TRACES. However, in the current version of DEJAVU, we found that the technique is, typically, less efficient than the combined DFS and compressed Schreier strategy (see Section 6.3.1). Hence, the current version of the preprocessor does *not* feature probing.

5.7 High-level Algorithm of the Preprocessor

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 non-canonical color refinement to produce an equitable coloring. The coloring remains equitable throughout the entire algorithm. We also continuously remove singletons. Beyond this, the implementation allows the user to freely specify a schedule for the various techniques.

The default schedule is as follows. We remove vertices of degree 0 and 1 (Section 5.4.1 and Section 5.4.3), apply edge flips (Section 5.3.2), and possibly repeat removing vertices of degree 0 or 1. Then, we remove true and false twins (Section 5.4.2). If some true or false twins were indeed removed, we restart the schedule from the beginning. Lastly, we apply the heuristics described for vertices of degree 2: first, checking for matchings (Section 5.4.4), then unique endpoints (Section 5.4.4), followed by edge flips (Section 5.5.1), and lastly the densification heuristic (Section 5.5.2).

The lemmas provided throughout this section show the correctness of the overall algorithm.

Chapter 6

The dejavu Algorithm

We now describe the main practical contribution of this thesis: the DEJAVU algorithm for symmetry detection. The solver is based on both the theoretical and practical insights gathered throughout the previous chapters.

The main ingredient of the solver is a Monte Carlo algorithm for symmetry detection. We argue correctness, i.e., error probability, and runtime of this algorithm. Notably, considering the runtime, the factor in the size of the IR tree reflects the portion of the search tree explored by the Monte Carlo strategy given in Chapter 3. Furthermore, we prove that the algorithm implicitly performs perfect automorphism pruning: the presence of automorphisms directly improves the runtime without any explicit pruning. The Monte Carlo algorithm is then extended with further techniques. The purpose of the chapter is to provide a thorough description of the solver and to argue the correctness of all further components. Furthermore, we argue the correctness of certain interactions between the components, whenever this is relevant. In the next chapter, benchmarks demonstrate the efficiency of the implementation.

Let us begin by discussing the overall design philosophy and strategy of DEJAVU.

6.1 Design Principles of the Solver

Ideally, one should not be required to pick and choose the right symmetry detection tool for an application. There should be a single tool that has adequate performance across all the different kinds of applications and graphs. The main goal of the DEJAVU solver is to create a tool that scales well on as many types of graphs as possible. *But how can this be achieved?*

The overarching assumption used in the creation of the solver is that computing the automorphism group of most graphs is easy, once tackled with the right kind of strategy. The solver tries to apply a carefully chosen sequence of procedures on the graph and the IR tree. If these procedures are not able to solve the graph within a given "cost budget", a new cell selector and target leaf are chosen in the hopes that they might reveal a more efficient strategy for solving the graph. The budget is increased in an exponential back-off scheme. In order to retain as much information as possible, when choosing a new cell selector and target leaf, an attempt is made to simplify the graph using the information gathered up to that point.

The most important component of the algorithm is the Monte Carlo traversal, similar to the one described for isomorphism testing in Chapter 3. Recall that the Monte Carlo strategy consists of random walks in the IR tree. Indeed, random walks in the IR tree

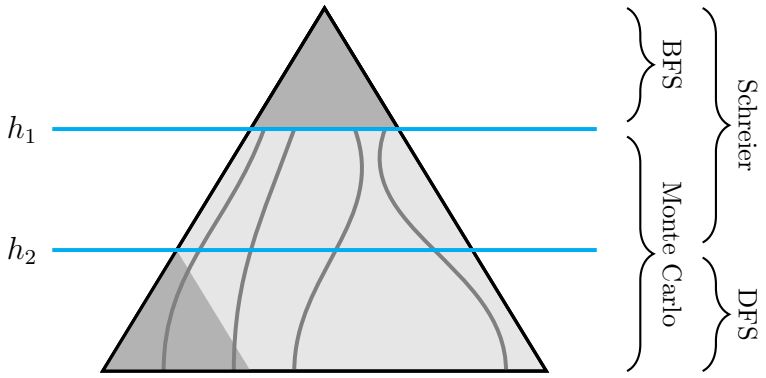


Figure 6.1: Relation between depth-first search, breadth-first search, Monte Carlo search, and the Schreier structure in DEJAVU. In the illustration, breadth-first IR search is performed from the root up until h_1. Depth-first IR search is performed from a leaf up to level h_2.

fulfill *two* purposes. Firstly, our theoretical analysis suggests the strategy is a near-optimal way of traversing IR trees in the worst case. But not only does it seem like a good choice for traversing the IR tree; random walks are also a natural tool to gauge the size and difficulty of the tree and its individual levels. The Monte Carlo traversal is facilitated by depth-first search and breadth-first search, as well as other optimization strategies, outlined below.

The underlying machinery of the solver is based on the efficient color refinement algorithm and other facilities described in Chapter 4. The preprocessor described in Chapter 5 is of course always applied before running the remainder of the algorithm.

The high-level procedure of DEJAVU is described in Algorithm 23. We give a description of the algorithm below, but more details on each of the steps can be found throughout this chapter.

Description of Algorithm 23. The algorithm begins by applying color refinement to the graph. Here, the color refinement is specifically a non-canonical one, meaning the partition is isomorphism-invariant but the ordering of the colors is not (see Section 4.4.4).

This is followed by applying the preprocessor techniques described in the previous section. The last preprocessor technique splits up the graph into non-uniform components (see Section 5.6.1). The remainder of the algorithm is applied to each non-uniform component separately (Line 6).

We then initialize a *budget* to 1, and enter the loop used for *restarts* (Line 10). Here, we first determine a cell selector and target leaf for the IR tree. Cell selectors are mostly chosen from an ensemble of cell selectors, but some limited randomization is applied, in particular when the budget is high. Then, we are ready to traverse the IR tree according to the selected cell selector and target leaf.

First, we perform a depth-first search (Line 12) on the IR tree, starting from the target leaf. We do so using matched vertex colorings (see Section 4.4.5) and the trace invariant (see Section 4.4.2). If the tree contains many sparse automorphisms, the hope is

that this solves the component very efficiently. The depth-first search gives up whenever it encounters a leaf that is non-equivalent to the target leaf. If it can not solve the component, it stops on some level h. If it did solve the component, we continue with the next component (Line 15).

If depth-first search was not successful, we continue by applying the Monte Carlo strategy, combined with breadth-first search. For this, we first need to initialize a Schreier structure with a base (the target leaf). A crucial fact is that we only need to initialize the Schreier structure up to level h. This is because depth-first search guarantees that we have found all the automorphisms starting from level h (i.e., generators of the stabilizer of the first h points). Hence, we only need to ensure that we find all the automorphisms up to level h.

Then, a heuristic decides whether we perform random walks in the IR tree and sift automorphisms into the Schreier structure towards the probabilistic abort criterion, or we perform breadth-first search in the hope of pruning nodes of the tree. We also compare the transversals of the Schreier structure to the color classes individualized on the walk to the target leaf, which essentially gives us an additional deterministic abort criterion. We continuously track the cost of these methods and compare it to our budget. The process is continued until one of the following is satisfied:

1. The budget is exceeded.
2. The deterministic abort criterion or probabilistic abort criterion are satisfied and we have solved the component.
3. Breadth-first search solves the component.

Each time the budget is exceeded, the algorithm doubles the budget. In this case, we first try to simplify the component using the information gathered so far (e.g., using automorphisms and breadth-first levels of the tree). Then, we go back to choosing a new cell selector and target leaf.

Figure 6.1 illustrates the relationship between depth-first search, breadth-first search, the Monte Carlo search, and the Schreier structure in the algorithm. In the remainder of this chapter, we explain each of the involved procedures in more detail.

6.2 Random Search and Breadth-First Search

The core routine of the DEJAVU algorithm is random walks in the IR tree combined with breadth-first search. We first describe the Monte Carlo algorithm for automorphism computation. We argue its correctness and runtime. In particular, we discuss why, at least theoretically, it does not require any form of additional automorphism pruning.

Then, we discuss breadth-first search and a crucial implementation strategy called "trace deviation sets". We carefully argue why the interplay of the breadth-first strategies and the Monte Carlo algorithm produces correct results.

Lastly, we discuss the heuristic of the solver that chooses between random search and breadth-first search.

Algorithm 23: High-level description of the DEJAVU algorithm.

```
1  function DEJAVU
       Input: ➢ graph G
              ➢ coloring π
              ➢ error probability 0 < ε ≤ 1
      Output: ≺ set S ⊆ Aut(G, π), with probability at least 1 − ε,
              ⟨S⟩ = Aut(G, π) holds
2      refine π using non-canonical color refinement;
3      // preprocess graph according to Section 5.7
4      preprocess (G, π);
5      // all subsequently detected symmetries are lifted accordingly
6      foreach non-uniform component (G′, π′) of (G, π) do
7          // we are only treating (G′, π′) now
8          budget ≔ 1;
9          // loop for restarts
10         while true do
11             make a new cell selector and target leaf;
12             perform limited depth-first search;
13             let h be level where depth-first IR search ended;
14             // did depth-first search solve this component?
15             if h = 0 then break;
16             initialize Schreier structure with h levels;
17             perform random IR search and/or breadth-first IR search until
                probabilistic criterion satisfied or budget exceeded;
18             // did we solve this component now?
19             if probabilistic criterion or deterministic criterion satisfied then
                break;
20             // if not, we exceeded our cost budget
21             inprocess component (G′, π′);
22             budget ≔ budget·2;
```

6.2.1 Monte Carlo Algorithm for Symmetry Detection

We begin by describing how a random walk in an IR tree can be computed. A key observation is that by choosing uniform, random walks through the tree we also get a uniform distribution of elements in the automorphism group.

Description of Algorithm 24. The algorithm applies the refinement to the input graph and then repeatedly chooses a vertex uniformly at random from the target cell chosen by the cell selector. The chosen vertex is then individualized. Starting from the initial coloring, it then keeps individualizing and refining until the coloring becomes discrete. It returns the coloring and the sequence of individualized vertices.

Runtime of Algorithm 24. We recall the fact that a path in the IR tree only incurs a cost of $\mathcal{O}((n+m)\log n)$ in color refinement (see Section 4.4.1). In the implementation, all the cell selector calls collectively run in linear time. Thus, if an appropriate cell selector is used, and a random element can be chosen in time $\mathcal{O}(1)$ (Line 9), a random walk in the IR tree can be computed in time $\mathcal{O}((n+m)\log n)$.

Correctness of Algorithm 24. Let τ' be an occurrence of a fixed target leaf τ. In this situation we call φ with $\varphi(\tau) = \tau'$ the *corresponding automorphism* with regard to τ'. Recall that Lemma 18 shows, that there is a unique occurrence of τ for every $\varphi \in \mathrm{Aut}(G)$. We are now ready to argue the correctness of the algorithm, which is that for any fixed leaf τ, the algorithm produces uniform random elements of the equivalence class of τ.

Lemma 76. *As a random variable, the output of Algorithm 24, which is a leaf in the search tree, is uniformly distributed within each equivalence class of leaves.*

Proof. There is a unique occurrence of τ for every automorphism (Lemma 18). Hence, it suffices to argue that the probability of finding each occurrence of τ through a random walk in the tree is equal. Assume that we are in a node ν of the search tree and let $\nu_1, \ldots, \nu_k$ be the children of ν. Let $\nu'_1, \ldots, \nu'_k$ be the children that correspond to the subtrees of ν that contain an occurrence of τ. Since we are sampling an element uniformly from $\nu_1, \ldots, \nu_k$ in Algorithm 24, each of these subtrees has the same probability of being chosen. Therefore, it suffices to argue that the chance of finding an occurrence of τ in each of $\nu'_1, \ldots, \nu'_k$ is equal. Since they all contain an occurrence of τ, they can all be mapped to each other using the corresponding automorphisms. But this immediately implies that all of these subtrees must be isomorphic (Lemma 14), showing the claim. □

Corollary 77. *Let τ be a fixed target leaf. Consider the distribution of outputs of Algorithm 24 under the condition that an occurrence of τ is computed. For such a given output τ' consider the automorphism φ with $\varphi(\tau) = \tau'$ corresponding to τ'. Then φ is uniformly distributed in* $\mathrm{Aut}(G)$.

Next, we turn to using random walks to compute the automorphism group of a given graph. The procedure is described in Algorithm 25.

Algorithm 24: Random walk of an IR tree.

```
1  function RandomWalk
       Input: ➢ graph G
              ➢ coloring π
      Output: ≺ a random leaf of the search tree
              ≺ individualized vertices
2      // initialize base to empty list
3      B := ϵ;
4      π := CRef(G, π, B);
5      C := Sel(G, π);
6      // perform random root-to-leaf walk in IR tree
7      while C ≠ ∅ do
8          // pick a random vertex of selected cell to individualize
9          v := RandomElement(C);
10         B := Bv;                                   // append v to base
11         // do IR
12         C := Sel(G, CRef(G, π, B));
13     return (π, B);
```

Description of Algorithm 25. The algorithm repeatedly samples automorphisms from the automorphism group using random walks (Algorithm 24). This is done by keeping all the randomly sampled leaves in a set L. The algorithm compares all subsequently sampled leaves to the leaves already contained in L. Then, it uses a probabilistic test based on Lemma 8 to terminate. When a certain number $d = \lceil -\log_2(\frac{\varepsilon}{2}) \rceil$ of *consecutively* sampled automorphisms turn out to be already covered by the previously found automorphisms (i.e., they sift successfully) the algorithm terminates. The initial value of d is linked to the guaranteed bound on the error probability of the algorithm ε that can be chosen by the user. To guarantee that the error bound is kept, when some but less than d consecutively found automorphisms sift successfully, the value of d is incremented. (This is loosely related to sequential testing in statistics. See also Section 2.7 of [103].)

Correctness of Algorithm 25. We argue that given a graph (G, π) and an error probability ε, the algorithm produces a generating set for the automorphism group of (G, π) with probability at least $1 - \varepsilon$.

First, observe that the discovered permutations are certified before being added to the group, which immediately ensures that all elements of the computed group are indeed automorphisms of the input graph. The algorithm can therefore only fail by not adding enough elements to the group.

We first argue that a random walk of the tree (as computed by Algorithm 24) either produces a new leaf of the tree, or a uniform random occurrence of a previously discvered leaf. Note that if no leaf equivalent to the output leaf τ' of Algorithm 24 has been seen before, τ' is added to the set of leaves, and no automorphism is computed. If a

Algorithm 25: Monte Carlo strategy for automorphism group computation.

```
1  function RandomAutomorphisms
       Input: ➢ graph G
              ➢ coloring π
              ➢ error probability 0 < ϵ ≤ 1
      Output: ≺ set S ⊆ Aut(G, π), with probability at least 1 − ϵ,
              ⟨S⟩ = Aut(G, π) holds
2      // error probability management
3      c := 0;
4      d := ⌈− log_2(ε/2)⌉;
5      // generating set S and set of leaves L
6      S := ∅;
7      L := ∅;
8      // base of the automorphism group
9      f_base := false;
10     B := ();
11     // probabilistic abort criterion
12     while c ≤ d do
13         // compute a random leaf
14         (τ', B') := RandomWalk((G, π));
15         // do we already have a base for the group?
16         if ¬f_base then
17             B := B';
18             f_base := true;
19         // check whether equivalent leaf has been seen before
20         f_leaf := false;
21         for ( τ ∈ L )
22             φ := τ · τ'^{-1};
23             if CheckAutomorphism((G, π), φ) then
24                 if ¬ Sift(S, B, φ) then c := c + 1;
25                 else
26                     if c > 0 then d := d + 1;
27                     c := 0
28                 f_leaf := true;
29                 break;
30         if ¬f_leaf then L := L ∪ {τ'} ;
31     return S;
```

leaf equivalent to τ' has been seen before, Corollary 77 ensures that the corresponding automorphism is a uniform random element of the automorphism group. This, in turn, enables us to use Lemma 8 to argue correctness as follows.

We terminate the algorithm when d consecutive uniform random elements of $\mathrm{Aut}(G, \pi)$ were successfully into the Schreier structure. As long as the sifting process fails and we add elements to the Schreier structure, we know that no error occurs and the process is not yet finished. We view the computation as a sequence of *tests* of the hypothesis that we are missing automorphisms. We define the beginning of a test to be right after sifting succeeds once (i.e., at the moment when c is set to 1 in an execution of Line 24). The probability that the test fails (i.e., that we do not abort the test early and instead increment c for d times in a row) is bounded by $(\frac{1}{2})^d$ (Lemma 8). In order to ensure a total error bound of ε for the algorithm, we require that the failure probabilities of the tests sum up to at most ε. For this it suffices that the i-th test fails with probability at most $\frac{\varepsilon}{2^i}$. The probability that the entire computation fails is then surely at most ε since

$$\sum_{i=1}^{\infty} \frac{\varepsilon}{2^i} \leq \varepsilon.$$

In order to satisfy this bound of $\frac{\varepsilon}{2^i}$, we increment d after each successful test. Initially, for the first test, we set $d_1 = \lceil -\log_2(\frac{\varepsilon}{2}) \rceil$ which ensures that $(\frac{1}{2})^d \leq \frac{\varepsilon}{2}$. Note that the value d_i for variable d used during the i-th test is then $d_i = d_1 + i - 1$, so $(\frac{1}{2})^{d_i} < \frac{\varepsilon}{2^i}$, as desired.

Runtime of Algorithm 25. We want to recover a runtime guarantee similar to the bounds of the Monte Carlo strategy of Chapter 3. In particular, we want to argue that the factor of the runtime in the size of the IR tree is only in the *square root* of the size of the tree.

Let us assume that the time for performing a random walk is bounded by $t_{\mathrm{walk}}(n)$, sifting an element of the group is bounded by $t_{\mathrm{sift}}(n)$, and checking whether an equivalent leaf exists by $t_{\mathrm{lookup}}(n)$. It is reasonable to assume that all of these procedures run in time that is polynomial in n, or more precisely, quasi-linear in n. We want to mention the lookup procedure can be implemented efficiently using a hash table and complete invariant, in order to locate candidates for equivalent leaves. Due to the nature of the Schreier structure, a trivial bound for filling the table is that it requires at most n^2 distinct automorphisms that sift unsuccessfully (see Section 2.1.3).

We claim that for a given constant probability ϵ, the algorithm runs in *expected* worst-case time

$$\mathcal{O}(n^2(t_{\mathrm{sift}}(n) + t_{\mathrm{lookup}}(n) + t_{\mathrm{walk}}(n))\sqrt{|T(G, \pi)|}).$$

Due to our assumptions above, we only need to argue that, in expectation, Algorithm 25 computes at most $\mathcal{O}(n^2\sqrt{|T(G, \pi)|})$ random leaves of the IR tree. We may assume that the algorithm does not terminate too early and that the Schreier structure is indeed filled to completion: earlier termination only leads to fewer computed leaves.

Let us first argue how many uniform random automorphisms it takes in expectation to fill the Schreier structure. Assume that the Schreier structure is not yet completely filled,

i.e., there exists an automorphism $\varphi \in \mathrm{Aut}(G, \pi)$ which fails to sift. Due to Lemma 8, in this case, a uniform random automorphism is not contained in the Schreier table with probability is at least $\frac{1}{2}$. Hence, $2n^2$ is a trivial upper bound on the expected number of uniform random automorphisms needed to fill the Schreier structure. A technicality remains: during the execution of the algorithm, we increase the value d whenever we sift automorphisms that are already in the Schreier structure, but the structure is not yet filled. Hence, more uniform automorphisms can be necessary to satisfy the probabilistic abort criterion. By assumption, at the beginning of the algorithm d is a constant. Let us give an upper bound on the expected number of automorphisms that are found but already in the Schreier structure, i.e., the expected value of d upon termination of the algorithm. Trivially, $d + 2n^2$ is an upper bound. Therefore, the expected number of automorphisms needed to satisfy the Schreier structure is bounded by $d + 4n^2$.

The question remains how many random walks of the individualization-refinement tree it takes to find each uniform random automorphisms. Clearly, as a random variable, this is independent of the number of automorphisms needed to satisfy the Schreier structure. Using similar arguments to Lemma 25, it follows that $c\sqrt{|T(G, \pi)|}$ suffice in expectation. (We also give a more fine-grained argument below.)

No automorphism pruning? Yes. We argue that for a given constant error probability ϵ, the algorithm runs in expected worst-case time

$$\mathcal{O}\left(n^2(t_{\mathrm{sift}}(n) + t_{\mathrm{lookup}}(n) + t_{\mathrm{walk}}(n))\sqrt{\frac{|T(G, \pi)|}{|\operatorname{Aut}(G, \pi)|}}\right).$$

Recall that the relevant factor in the size of the IR tree matches our observation of Lemma 22.

It suffices to argue that the likelihood of finding an occurrence in the search tree through random walks is amplified by the size of the automorphism group. Let (G, π) be a graph and $\tau \in L(T(G, \pi))$. In $T(G, \pi)$, there are $|\operatorname{Aut}(G, \pi)|$ occurrences of τ (Lemma 18). Let p be the probability of finding the node τ through a random walk of $T(G, \pi)$. But due to isomorphism invariance of $T(G, \pi)$ (Lemma 14), the probability of finding a specific occurrence τ' of τ is also p. Hence, the probability to find *any* occurrence is $|\operatorname{Aut}(G, \pi)|p$. We again assume uniformity, i.e., that the probability of finding a leaf is uniform across all leaves. If probabilities are non-uniform, the chance of finding leaves repeatedly increases (see [84]). In our specific case, the probability of finding an automorphism in L after i *distinct* leaves have been found is therefore at least

$$\frac{|\operatorname{Aut}(G, \pi)| \cdot i}{|T(G, \pi)|}.$$

There, after

$$\sqrt{\frac{|T(G, \pi)|}{|\operatorname{Aut}(G, \pi)|}}$$

random walks have been computed, either we have already found an automorphism, or all of the walks resulted in distinct leaves. Hence, we may assume the latter. The probability for finding an automorphism in $T(G, \pi)$ using a random walk is then at least

$$\frac{\sqrt{|T(G,\pi)|/|\operatorname{Aut}(G,\pi)|}}{|T(G,\pi)|/|\operatorname{Aut}(G,\pi)|} = \frac{1}{\sqrt{|T(G,\pi)|/|\operatorname{Aut}(G,\pi)|}}.$$

Using similar arguments to Lemma 25, this suffices to show that there is a fixed constant c, such that in expectation, the algorithm finds an automorphism after at most

$$c\sqrt{\frac{|T(G,\pi)|}{|\operatorname{Aut}(G,\pi)|}}$$

random walks.

Overall, this shows that IR trees are implicitly pruned using automorphisms: isomorphic copies of leaves actively contribute towards termination. In particular, in conjunction with Lemma 22, we can see that the algorithm exploits *all* automorphisms.

No automorphisms? Probably. Observe that the error of Algorithm 25 is *one-sided*. The algorithm may fail to discover symmetries of the input graph. However, $\langle S \rangle \subseteq \operatorname{Aut}(G, \pi)$ always holds.

I've just been in this place before. Termination of the algorithm hinges on seeing already explored leaves as well as already generated automorphisms again – a déjà-vu. The correctness of the algorithm depends on the fact that we are probing automorphisms uniformly from the group. In the next section, we introduce further techniques to prune the search tree. When we do so, we always make sure to do this in a manner that still enables us to probe uniformly after the pruning. Ensuring this suffices to retain the correct behavior of the algorithm.

6.2.2 Breadth-first Search with Trace Deviation

Combined with the Monte Carlo algorithm of the previous section, DEJAVU also performs a breadth-first search of the IR tree. The reason is that if nodes can be pruned using invariant pruning early in the tree, meaning close to the root, then random walks tend to fail early, and often. However, whenever we prune the search tree, and we want to apply Algorithm 25 again, we need to ensure that we prune the tree *uniformly*. Figure 6.2 illustrates the issue. Fortunately, for breadth-first search, pruning always occurs uniformly.

The algorithm for breadth-first search going from level k to $k + 1$ computes the corresponding node ν at level $k + 1$ that leads to the target leaf τ. It then takes all the remaining nodes from k, computes all their children at level $k + 1$. Let ν' be one of these children. Let Inv be an invariant (see Section 2.3.4, the precise invariant we use is described further below). If $\text{Inv}(\nu) \neq \text{Inv}(\nu')$, ν' is pruned.

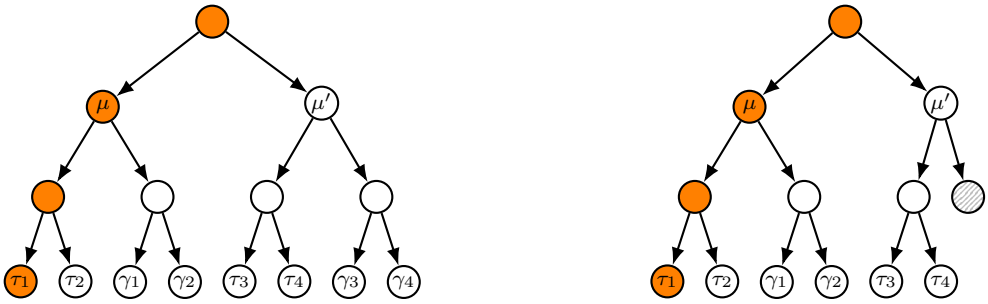


Figure 6.2: Example search tree illustrating non-uniform pruning. A search tree before (left) and after (right) pruning is shown. Orange nodes indicate the path of the target leaf τ_1. Leaves τ_i indicate occurrences of τ_1, γ_i indicates occurrences of γ_1. In the pruned tree, an occurrence of τ is found more likely in the subtree of μ' than μ (through random walks, assuming the immediate path to the pruned node is not taken).

Probing after a breadth-first traversal of an entire level has been performed can equivalently be characterized as starting random walks from the remaining nodes of the IR tree at a given level. We probe from a level k by choosing uniformly at random a node ν' of the IR tree with $|\nu'| = k$ that satisfies $\mathrm{Inv}(\nu') = \mathrm{Inv}(\nu)$. Here, ν is the prefix of length k of the vertex sequence corresponding to the target leaf. Using the argument of Lemma 76 again, the trees rooted in prefixes that contain some occurrence of the target leaf are isomorphic (Lemma 14). Therefore, the probability of finding an occurrence of the target leaf is the same in every such subtree. If we apply no other form of pruning, the entire process samples automorphisms with a uniform distribution.

Deviation Invariant. During breadth-first traversal, we keep a so-called *trace deviation set*. The idea of this pruning technique is related to the special automorphism algorithm of Traces (see Section 2.4.4) and the failure sets of bliss (see Section 2.4.3). We present the idea in the following.

We first define a new node invariant, which we call the *deviation value* $\mathrm{Dev}_{\mathrm{Inv}} \colon V^* \to \mathbb{N}^2 \cup \{\bot\}$. Consider a fixed invariant $\mathrm{Inv}(\tau)$, which, for our purposes, will be the trace invariant of the target leaf τ (see Section 4.4.2). The deviation value $\mathrm{Dev}_{\mathrm{Inv}}(\nu)$ for a node ν is then defined as a tuple of the first position and the corresponding value in the trace $\mathrm{Inv}(\nu)$ that is different from $\mathrm{Inv}(\tau)$. If there are no differences, we set the deviation value to $\bot$, denoting "no deviation". Since the deviation value is a function of the trace invariant computed until an isomorphism invariant point, it is naturally invariant under isomorphism.

Consider a node $\mu \in T(G, \pi)$ in the IR tree. The crucial observation is that in our algorithm, we can also use the set of deviation values of its children as an invariant for μ itself. Assume $\nu_1, \ldots, \nu_k$ are children of μ and none of the subtrees rooted in the children has been pruned through invariant pruning. Then,

$$D(\mu) \coloneqq \{\mathrm{Dev}_{\mathrm{Inv}}(\nu_1), \ldots, \mathrm{Dev}_{\mathrm{Inv}}(\nu_k)\},$$

the trace deviation set of μ, can be used as an invariant for μ: we claim that for any other node μ' with children $\nu'_1, \ldots, \nu'_k$,

$$D(\mu) = \{\mathrm{Dev}_{\mathrm{Inv}}(\nu_1), \ldots, \mathrm{Dev}_{\mathrm{Inv}}(\nu_k)\} = \{\mathrm{Dev}_{\mathrm{Inv}}(\nu'_1), \ldots, \mathrm{Dev}_{\mathrm{Inv}}(\nu'_k)\} = D(\mu')$$

must hold whenever μ and μ' are isomorphic. If no pruning has taken place, this is easy to see since the branches are isomorphic by assumption, immediately implying that branches must contain the same invariant values.

When advancing in a breadth-first manner, the aforementioned requirements are guaranteed to be satisfied: no invariant pruning has taken place on the level that is currently being pruned. Furthermore, while computing the level, the set of deviation values is automatically calculated anyway: whenever we find out that a node ν below μ deviates from the desired invariant $\mathrm{Inv}(\nu)$ and should be pruned, we already have enough information to derive $\mathrm{Dev}_{\mathrm{Inv}}(\nu)$.

Deviation Pruning. These observations are specifically exploited as follows: first, all children of the base node τ' (which belongs to the path on the way to the target leaf τ) are computed. If nodes deviate from the trace, their deviation values are recorded into a set, i.e., we calculate the trace deviation set $D(\tau')$. The idea is that if a node is (supposedly) isomorphic to the base node τ', then, for its (supposedly) isomorphic children, it must deviate from the trace in the same manner at the same position. Hence, for all other parent nodes μ, we also keep track of $D(\mu)$ when calculating their children. Whenever we discover a new element of $D(\mu)$, we check whether the equality of sets $D(\tau') = D(\mu)$ can still be satisfied. If not, μ can be pruned immediately without the necessity to calculate all of its children.

We collect the fact that the Monte Carlo strategy can be applied starting from a breadth-first level pruned with trace deviation. The lemma below suffices.

Lemma 78. *Let $T'(G, \pi)$ be an IR tree that was pruned with deviation pruning on a level k of breadth-first search, with target leaf τ. For any node $\mu \in T(G, \pi)$, the pruned tree $T'(G, \pi)$ either contains all nodes $\mu^{\mathrm{Aut}(G,\pi)}$, or none of them.*

Proof. If μ' is a node on level k' with $k' < k$, the statement follows immediately.

Let μ be any node, and μ^φ for $\varphi \in \mathrm{Aut}(G, \pi)$ be any occurence of μ in $T(G, \pi)$. If a node μ' at level k contains μ, then μ'^φ contains μ^φ. Each child ν of μ' has a corresponding isomorphic counterpart ν^φ in μ'^φ. Due to isomorphism-invariance of the invariant Inv, ν and ν^φ must have the same value, i.e., $\mathrm{Inv}(\nu) = \mathrm{Inv}(\nu^\varphi)$. It follows that

$$D(\mu') = \{\mathrm{Dev}_{\mathrm{Inv}}(\nu_1), \ldots, \mathrm{Dev}_{\mathrm{Inv}}(\nu_k)\} = \{\mathrm{Dev}_{\mathrm{Inv}}(\nu_1^\varphi), \ldots, \mathrm{Dev}_{\mathrm{Inv}}(\nu_k^\varphi)\} = D(\mu'^\varphi).$$

Hence, either both μ' and μ'^φ are pruned, or neither of them are pruned.

The above argument also holds whenever $\mu' = \mu$. □

Note that the above is enough to ensure that we can prove a result similar to Lemma 76 for random walks on the pruned tree $T'(G, \pi)$: again, we can think of starting Algorithm 24 from a random node at level k of $T'(G, \pi)$.

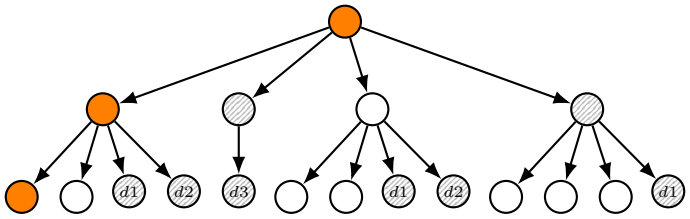


Figure 6.3: Potential search tree traversed when using trace deviation sets. Orange indicates base nodes, and gray indicates pruned nodes. Children of pruned nodes are of course eventually pruned as well.

For example, assume that we calculated deviation values $\{d_1, d_2\}$ for the base node (illustrated in Figure 6.3). From the previous discussion, it follows that we can immediately prune all nodes that produce a value other than $\{d_1, d_2\}$. We can also prune nodes that do not produce all of the deviation values. If for example a value $d_3 \notin \{d_1, d_2\}$ is encountered, the parent node can immediately be removed from the tree.

Practical Aspects. A crucial point is that pruning through trace deviation sets has negligible cost: children of the base node always have to be computed, and the trace deviation does not require more calculation than is done for that particular node anyway. We are still able to fully use the early-out capabilities of the trace invariant.

In the implementation a slight variation of the above technique is used: to distinguish deviation values further, it is sometimes beneficial to not use the early-out immediately. Instead, for a fixed constant k, color refinement is continued past the deviation for k more cells, accumulating more information for the deviation value. The trade-off is as follows: if k becomes larger, the early-out in color refinement is taken later, but deviation values become more distinct. In practice, this trades per-node cost for the number of nodes in the search tree. However, in our experiments we observed that even for small k, node reduction can be substantial – while not increasing per-node cost by a significant amount. The value for k used in practice is determined by a heuristic, which takes into account where random walks of Algorithm 25 deviated from the trace.

Implementation Pitfall. Typically, breadth-first search is carried out on a node-by-node basis. The peculiarity using trace deviation pruning is that it is possible to compute a new node ν of the IR tree that is not immediately pruned using invariant pruning. However, after computing one of its siblings ν' (both ν and ν' have the common parent μ), it might become necessary to prune ν (see Figure 6.3).

The reason is that ν might not be pruned using invariant pruning, but ν' might indicate a trace deviation of their common parent μ. In order to achieve uniformity of the resulting tree, it is crucial to remove ν as well.

Comparison to Related Techniques. The trace deviation technique is related to a family of pruning techniques that make use of the isomorphism-invariance of the IR tree itself.

The first related technique is the special automorphism algorithm used in TRACES (see Section 2.4.4), or in a sense, the Las Vegas algorithm (see Section 3.2.2). Specifically, these strategies are related in the case when nodes are not isomorphic. Both try to exploit the technique of first handling all children for one node to then compute fewer children for the other nodes. The techniques are however not immediately comparable: on the one hand, the technique of TRACES is more generic since it can also gain benefits on isomorphic branches. Also, it *guarantees* that only a single leaf for each node has to be explored after the base node has been fully computed. On the other hand, the technique described here is, in a sense, more generic since it can be applied – with negligible cost – on all levels of the breadth-first search. Another related technique are the so-called *failure sets* of BLISS (see Section 2.4.3).

Automorphism Pruning. An earlier version of DEJAVU contained automorphism pruning while performing breadth-first search. This comes with the additional challenge of ensuring that the automorphisms probed from the tree are still uniform. The solution as detailed in [5] adds weights to the tree, according to how many isomorphic nodes each node in the tree represents. These weights are then accounted for when probing in the tree.

The version of DEJAVU described in this thesis never combines the use of automorphism pruning and probing. One reason is that the introduction of the weights adds severe implementation complexity. Another reason is that the cases in which significant speedup was observed were mostly due to cases in which automorphisms are discovered *during* breadth-first search, i.e., on the last level of the IR tree. For this specific case, the current version also has a limited version of automorphism pruning. Here, the algorithm is guaranteed to immediately terminate after breadth-first search is finished, i.e., that no further probing is needed.

6.2.3 Choosing between Monte Carlo and Breadth-first Search

We have a heuristic that continuously decides whether to compute random walks or another level of breadth-first search. This heuristic is based on a cost estimation, which we describe in the following.

Consider the following question: given a constant c, is it more expensive to compute c random walks of the IR tree, or the next level of breadth-first search?

We can use previously computed random walks as *samples* for the next breadth-first level. Given enough samples k, we can compute a good estimate on the number of nodes on the next breadth-first level that remain under invariant pruning. The idea is that we are continuously sampling random walks, which can detect whether we will prune nodes via invariant pruning. Let n_l denote the number of breadth-first nodes remaining at level l. We assume all nodes have the same number of d children. (In practice, this is guaranteed by the fact that the trace invariant would immediately prune nodes with differing selected cells.) This means there are precisely dn_l children at level $l+1$. We assume that we have k uniform samples of the dn_l children, meaning we know for k uniform random children on level $l+1$ whether they are pruned or not. Let k_{prune} denote

the number of samples that correspond to pruned nodes. Our estimate for n_{l+1} is then $\frac{k_{\text{prune}}}{k} d n_l$.

In practice, we try to make even more fine-grained estimations. We incorporate the following variables:

1. The expected cost of a random walk versus the expected cost for each breadth-first node, which is estimated by considering the length of the trace used for each.
2. The expected cost of reversing the refinement, i.e., resetting the entire coloring for random walks versus using reversible refinements (see Section 4.4.3) for breadth-first search.
3. The effects of trace deviation pruning.

This gives us a cost estimate of whether c random walks or the next breadth-first level should be cheaper. However, the decision heuristic does not simply choose the cheaper of these two options. Intuitively, this is also not what it should do: what we want to choose, is the method that has the greatest *impact* on finishing the computation. Unfortunately, this does not seem to be something that we can easily estimate in a rigorous manner.

In general, we apply a bias coefficient that prefers random walks over breadth-first search, since this both improves our estimates and potentially solves the graph immediately, whereas breadth-first search usually does not. We apply different biases to the cost estimation in certain situations where the effectiveness seems more clear:

1. If we are already regularly finding equivalent leaves, we strongly prefer random walks to finish the search.
2. If breadth-first search *will* prune nodes on the next level, we apply a bias to prefer it. In particular, we apply a strong bias whenever the estimate for n_{l+1} is smaller than n_l, i.e., whenever the pruned size of the next level is smaller than the size of the current level.

6.3 Random Search and Depth-first Search

As described in Algorithm 23, before applying the Monte Carlo algorithm, DEJAVU first runs a limited depth-first search of the IR tree. In this section, we discuss the reasoning behind the depth-first search, the routine itself, and further interactions with the random automorphism search.

6.3.1 Limited Depth-First Search

At this point, the reader might be wondering: *why depth-first search?* The primary motivation is quite simple, however: to reduce the size of the Schreier structure. Indeed, for large practical graphs, the Schreier structure can quickly grow quite large. The previous version of DEJAVU [5] often ran out of memory on these graphs. The same holds true for NAUTY (when using the Schreier-Sims algorithm) and TRACES.

In turn, we now employ *three* key strategies to reduce the size of the Schreier structure:

1. Running a depth-first search prior to the Monte Carlo search, which reduces the length of the base in the Schreier structure (explained in this section).

2. A domain compression, as explained in detail in Section 6.3.2.

3. Generators and transversals of the Schreier structure are stored in a sparse manner.

Depth-first search is therefore aimed specifically at graphs which exhibit a lot of symmetry, and in turn produce large Schreier structures. However, in such cases, depth-first search can even be a beneficial strategy to begin with: as explained in detail in Section 4.4.5, matched vertex colorings can be used to efficiently determine automorphisms with small support. (Essentially, we use a very similar strategy as applied in the depth-first search of SAUCY.) On the other hand, often few random automorphisms suffice to generate the entire group – however, random automorphisms tend to have a large support, and filling the Schreier structure with random automorphisms comes at a cost. Hence, there is a trade-off.

Coming back to depth-first search, the goal is clear: we only want to discover automorphisms *efficiently* prior to our random search. In particular, if the graph seems difficult and requires a more extensive search due to non-equivalent leaves, we immediately stop the depth-first search. The sole purpose of depth-first search is to efficiently reduce the size of the Schreier structure, whereas difficult graphs ought to be handled by our other routines. Algorithm 26 and Algorithm 27 contain a sketch of the depth-first search used in DEJAVU.

Description of Algorithm 26. The algorithm begins at the target leaf. It backtracks one level from the target leaf (Line 7), and attempts to determine that all vertices of the respective color class (Line 11) are in fact in the same orbit of the base vertex v_b (in the pointwise stabilizer $\mathrm{Aut}(G,\pi)_{(\tau')}$). (In other words, when depth-first search is at level l, base vertex means the corresponding vertex on the way to the target leaf at level l.) For this, an orbit partition is maintained and consulted for each of the candidate vertices (Line 15). If this check fails, the algorithm tries to find an automorphism mapping v_b to v. This procedure is described in Algorithm 27. If the procedure returns an automorphism, the automorphism is added to the orbit partition (Line 20) and we continue the search (see Section 2.2.6 for the orbit algorithm used). Otherwise, we would have to backtrack due to non-equivalent leaves – in which case, as outlined above, the process is halted. The algorithm returns the level up to which it determined all the automorphisms (see Line 24 and Line 26). Whenever a level of depth-first search is completed, the algorithm backtracks to the next level (see Line 25).

Description of Algorithm 27. The main idea of this procedure is to exploit matched vertex colorings (see Section 4.4.5). Recall that once the colorings are matched, we only need to check whether the particular permutation which is determined by the singletons of the colorings is an automorphism of the graph (see Lemma 67). Instead of using a given, fixed cell selector, we individualize and refine two colorings in lock-step. We always choose a non-trivial color that is not yet matched in the two colorings (Line 6). Once the colorings

are matched (observe that when colorings are discrete, they are matched), we check whether the corresponding permutation is an automorphism of the graph (Line 17). If it is not an automorphism, or the colorings become non-isomorphic at any point (Line 15), the procedure would have to backtrack due to a non-equivalent leaf. Therefore, instead, the algorithm simply terminates without producing a result.

Correctness of Algorithm 26. The algorithm describes a limited version of a standard depth-first search of the IR tree. We record the basic fact that if the depth-first search returns a level h, then the automorphisms returned by depth-first search generate at least the corresponding pointwise stabilizer of $\mathrm{Aut}(G, \pi)$.

Corollary 79. *Let (h, S) be the output of Algorithm 26 on a graph (G, π) and target leaf τ. Let τ' be prefix of length h of τ. Then, $\mathrm{Aut}_{(\tau')} \subseteq \langle S \rangle$ holds.*

Implementation of Algorithm 26 and Algorithm 27. In order to ease the presentation, the algorithm as described is missing a few crucial optimizations. Let us collect some of them, here. First, of course colorings are not always recomputed: the implementation only ever backtracks using reversible refinements (see Section 4.4.3), and moves forward in the IR tree by individualizing single vertices (see Section 4.4.1). Furthermore, in Algorithm 27, whether the two colorings are matched is handled dynamically during color refinement itself (see description in Section 4.4.5). Automorphisms are actually not stored in a generating set, but output immediately to the user using a single dense-sparse data structure (see Section 2.2.3). In particular, this enables the efficient computation of many automorphisms with small support. Lastly, invariants are used to determine more quickly whether there is any hope of finding automorphisms, as is described below.

Limited Invariant Pruning. Algorithm 26 and Algorithm 27 continuously apply and check a trace invariant (see Section 4.4.2), in order to more quickly terminate search. However, let us consider the particular case in which Algorithm 27 rejects τ and $\tau' v$ on the *first* individualization. In this case, no further search is required: we have successfully determined that τ and $\tau' v$ (see Line 17) are indeed *not* in the same orbit of the pointwise stabilizer. In particular, this is true if τ and $\tau' v$ lead to different invariants. The same holds true whenever the colorings of τ and $\tau' v$ are matched, but do not result in a valid automorphism (see Lemma 67). Hence, in these cases, the algorithm continues with the search.

Trace Cost Termination. As was discussed previously, there is a trade-off between discovering automorphisms using depth-first search, and discovering them using random search while filling a Schreier structure. In particular, we expect random search using the Schreier structure to require fewer total automorphisms. The random search does however lack the matched vertex coloring technique, and automorphisms tend to have a larger support. We use a heuristic to stop depth-first search in favor of random search.

We do so by keeping track of the average "trace cost" per automorphism. Assume that the trace invariant for the target leaf τ has length k. For each automorphism in

Algorithm 26: Depth-first search of IR tree.

```
1  function DFS
       Input: ➢ graph G
              ➢ coloring π
              ➢ target leaf τ
      Output: ≺ level h
              ≺ generators S with Aut(G, π)_(τ') ⊆ ⟨S⟩, where τ' is τ-prefix of
                length h
2      initialize trivial orbit partition Δ, where each vertex is in its own orbit;
3      S := ∅;
4      // as long as there are levels to backtrack
5      while |τ| ≥ 1 do
6          // backtrack one base point in τ
7          τ' := τ_1 ... τ_{|τ|-1};
8          // which vertex was individualized on the way to τ?
9          v_b := τ_{|τ|};
10         // which color was individualized on the way to τ?
11         C := Sel(G, CRef(G, π, τ'));
12         // consider all vertices of C
13         foreach v ∈ C do
14             // if we already know v and v_b are in the same orbit,
                  continue
15             if v and v_b are in same orbit of Δ then continue ;
16             // otherwise, look for an automorphism mapping v to v_b
17             φ := MatchRecurse(G, π, τ, τ'v);
18             // if we found an automorphism, continue, otherwise
                  terminate
19             if φ ≠ ⊥ then
20                 extend orbit partition Δ with φ;
21                 S := S ∪ {φ};
22                 output φ;
23             else
24                 return (S, |τ|)
25         τ := τ';
26     return (S, |τ|)
```

Algorithm 27: Search for automorphisms between two nodes in IR tree.

```
1  function MatchRecurse
       Input: ➢ graph G
              ➢ coloring π
              ➢ list of vertices ν
              ➢ list of vertices μ
       Output: ≺ automorphism φ ∈ Aut(G, π) mapping ν to μ, or ⊥
2      π_ν ≔ CRef(G, π, ν);
3      π_μ ≔ CRef(G, π, μ);
4      while (π_ν, π_μ) not matched do
5          // find a non-matching color class
6          c := non-trivial color of π_ν, π_μ with π_ν^{-1}(c) ≠ π_μ^{-1}(c);
7          // find vertices which cause color classes to not be matched
8          v_ν ≔ v ∈ π_ν^{-1}(c) and v ∉ π_μ^{-1}(c);
9          v_μ ≔ v ∈ π_μ^{-1}(c) and v ∉ π_ν^{-1}(c);
10         ν ≔ νv_ν;
11         μ ≔ μv_μ;
12         // now individualize these vertices
13         π_ν ≔ CRef(G, π, ν);
14         π_μ ≔ CRef(G, π, μ);
15         if π_ν and π_μ are non-isomorphic then
16             return ⊥
17     if CheckAutomorphism((G, π), φ_{π_ν,π_μ}) then
18         return φ_{π_ν,π_μ}
19     else
20         return ⊥
```

depth-first search, we keep track of how much of the trace we needed to compute in order to detect the automorphism. Let k' be the length of the recomputed trace. We then continuously track the ratio $\frac{k'}{k}$ for detected automorphisms. (The implementation averages the ratio of multiple recent automorphisms.) In particular, this means if τ' in Algorithm 26 is still fairly close to a leaf, the ratio will be small. Furthermore, if colorings in Algorithm 27 match within few individualizations, the ratio will be small as well. Once the ratio exceeds a certain threshold, depth-first search is terminated. The typical threshold used in the implementation is 0.25, but larger values are used if the expected absolute size of the Schreier structure is large.

6.3.2 Monte Carlo Algorithm and Schreier-Sims, Revisited

We revisit our Monte Carlo algorithm of Section 6.2.1. We describe the interaction with the depth-first search, as well as further optimizations to the approach.

Specifically, we assume that we have successfully performed depth-first search using the target leaf τ up to level h. Instead of just using the first encountered base, we specifically choose the base of the Schreier structure used in Algorithm 25 to be τ. This will aid in some of the optimizations described below.

Partial Base. Let $h' = |\tau|$ and assume that $h \leq h'$. Furthermore, we let $\tau' = \tau_1 \dots \tau_h$ denote the prefix of h vertices of τ.

We observe that depth-first search ensures that we have found all the elements of the pointwise stabilizer $\mathrm{Aut}(G, \pi)_{(\tau')}$ (see Corollary 79). Hence, due to the orbit-stabilizer theorem, it suffices now to find automorphisms $S \subseteq \mathrm{Aut}(G, \pi)$ such that

$$\langle S, \mathrm{Aut}(G, \pi)_{(\tau')} \rangle = \mathrm{Aut}(G, \pi).$$

We observe that using the Schreier structure with the partial base τ' ensures to find such a set S. In particular, this means we sift an automorphism only according to the partial base τ'. If the automorphism successfully sifts through all the levels of τ', but is not the identity, we deem the sift successful. The Schreier structure is not extended (see Algorithm 1).

Deterministic Termination. Again, we let $\tau' \in T(G, \pi)$ denote any prefix of the target leaf τ in $T(G, \pi)$. We include that case that $\tau' = \tau$, but exclude $\tau' = \epsilon$. We let $\tau'' v = \tau'$ for $v \in C$, where $C = \mathrm{Sel}(G, \mathrm{CRef}(G, \pi, \tau''))$. Furthermore, we let $\pi(\tau'')$ denote the corresponding coloring of τ'', and $C(\tau'')$ the set of vertices contained in the color selected at node τ''.

It follows readily from Corollary 15, that the orbit $v^{\mathrm{Aut}(G,\pi)_{(\tau'')}}$ is a subset of the vertices in C. In other words, the size of the color class of v in the IR tree is an *upper bound* for the size of its orbit in the pointwise stabilizer $\mathrm{Aut}(G, \pi)_{(\tau'')}$. (Recall that the orbit of v in the pointwise stabilizer corresponds to the transversal in the corresponding level of the Schreier table.)

In particular, this implies that if $|C|$ equals the size of the transversal of v stored in the Schreier table, we know that the transversal can not be extended any further. If this

is true on *all* levels of the Schreier table, we know that we have filled the Schreier table to completion, and thus we have found all automorphisms of the graph.

Hence, this can be used to terminate Algorithm 25 immediately, even guaranteeing that all automorphisms have been found. We refer to this as the *deterministic abort criterion.*

Since color refinement is quite effective in determining the orbit partition for many graph classes, the deterministic abort criterion is often applicable. Indeed, for practical graphs, DEJAVU rarely invokes its probabilistic abort criterion. In particular, for Tinhofer graphs [11], DEJAVU is *guaranteed to terminate deterministically.*

Domain Compression. A crucial observation is that we only ever use the Schreier structure to check whether a given automorphism is in the group or not. In particular, we do not need to output elements from the Schreier structure itself – these elements are generated by the automorphisms found by random search.

This observation gives rise to the following idea: we may be able to maintain the Schreier structure on a smaller domain $V' \subseteq V(G)$ that still suffices to perform the check. Indeed, if such a smaller domain V' already fully determines the automorphisms, then there is no need to store the other vertices.

The idea used in the following is that the target leaf τ is a complete base of the automorphism group (Lemma 17). Intuitively, this means that however the points of τ are mapped uniquely identifies an automorphism.

Formally, given a graph (G, π) and target leaf τ we define the *compressed domain* $V' \subseteq V(G)$ as follows: For all $v \in V(G)$, $v \in V'$ holds if and only if there is a $v' \in \tau$ such that $v \in \pi^{-1}(\pi(v'))$. Essentially, V' is guaranteed to contain all the points to which points of τ could be mapped under automorphisms. We show that a Schreier structure on the domain V' suffices for our purposes.

Lemma 80. *For a graph (G, π) and leaf $\tau \in T(G, \pi)$, let $V' \subseteq V(G)$ be the compressed domain of τ. Then, for all automorphisms $\varphi \in \mathrm{Aut}(G, \pi)$ and subsets of automorphisms $S \subseteq \mathrm{Aut}(G, \pi)$, the following holds:*

$$\varphi|_{V'} \in \langle S|_{V'} \rangle \text{ if and only if } \varphi \in \langle S \rangle.$$

Proof. First, recall that τ is a complete base of $\mathrm{Aut}(G, \pi)$ (Lemma 17). This means the pointwise stabilizer $\mathrm{Aut}(G, \pi)_{(\tau)}$ is trivial.

Let $\varphi_1 \in \mathrm{Aut}(G, \pi)$ and assume towards a contradiction there is a $\varphi_2 \in \mathrm{Aut}(G, \pi)$, such that $\varphi_1 \neq \varphi_2$ and $\varphi_1|_{V'} = \varphi_2|_{V'}$.

We let $W := \{v \mid \varphi \in \mathrm{Aut}(G, \pi), v \in \tau^{\varphi}\}$ denote all the vertices to which vertices of τ can be mapped to using automorphisms of (G, π). The colors of π, by definition, are an over-approximation of the orbit partition of $\mathrm{Aut}(G, \pi)$. Hence, V' is guaranteed to contain W, i.e., $W \subseteq V'$ holds.

Since $\varphi_1|_{V'} = \varphi_2|_{V'}$, we know that $\varphi_1(\tau) = \varphi_2(\tau)$ holds. In particular, we know that $\varphi_1\varphi_2^{-1}(\tau) = \tau$ and therefore $\varphi_1\varphi_2^{-1} \in \mathrm{Aut}(G, \pi)_{(\tau)}$ hold. However, since $\varphi_1 \neq \varphi_2$, we know that $\varphi_1\varphi_2^{-1} \neq \mathrm{id}$. This contradicts the fact that $\mathrm{Aut}(G, \pi)_{(\tau)}$ is trivial. □

This shows that we can define our Schreier structure on the compressed domain V' instead of the original one. (Observe that the arguments above also naturally hold for the case when we are considering a partial base.)

The effectiveness of the technique of course heavily depends on whether the group actually contains parts not required to understand the entire group action. Another important choice is that of the base τ.

In the implementation, we apply the technique whenever the compression ratio, i.e., the quotient $|V'|/|V|$, is sufficiently small. Then, we create a map $p : V(G) \to \{1, \ldots, |V'|, \perp\}$ that either maps a given vertex to a point in $1, \ldots, |V'|$ interval, or signifies that the vertex is not needed in the Schreier domain using $\perp$. The Schreier structure and algorithm can then just be used as usual on the domain $\{1, \ldots, |V'|\}$.

Base-aligned Search. Whenever we run the Monte Carlo algorithm from the root of the tree, we apply another optimization. Observe that this is the case whenever we have not run any breadth-first search yet. We perform a so-called *base-aligned* search.

As the name suggests, base-aligned search initiates random walks from a *base point*, i.e., a node that corresponds to a prefix of τ. The base point is advanced whenever it is detected that the target cell of the current base point is equal to the orbit in the Schreier structure. As argued above, whenever this is the case the orbit on this level can not be extended any further. Hence, the sampled automorphisms are still uniform.

Since base-aligned search finds *all* automorphisms of graphs with easily structured IR trees, e.g., Tinhofer graphs, optimizations for these cases turn out to be highly beneficial. We found that preferring base points whenever possible, i.e., whenever it is already known that the randomly chosen point is in the same orbit as the base point, can drastically reduce sifting cost.

6.4 Restarts and Strategy Sampling

As outlined in Algorithm 23, the solver features the use of restarts. On each restart, the solver potentially alters the cell selector used and the target leaf.

We first describe the different strategies employed by the solver. Then, we describe the mechanisms and heuristics surrounding restarts: sometimes, restarts are mitigated, whereas other times a strategy is immediately discarded and the solver restarts again.

6.4.1 Cell Selectors

Below, we give a description for the cell selectors of DEJAVU. The cell selector is varied upon restart. Furthermore, all cell selectors are used in the following way: the target leaf is computed using the selector as described. For efficiency reasons, in subsequent branches of the IR tree, the solver simply chooses the color that was chosen on the branch of the target leaf. If this is *not* a non-trivial color class in the subsequent branch, then the first non-trivial color is chosen.

The ensemble of cell selectors used by DEJAVU is as follows.

Connected Colors. The first cell selector chooses a color for which the product of the number of adjacent colors and the size of the color class itself is largest. Formally, we choose a color class C which maximizes

$$|C||\pi(N(C))|.$$

The cell selector augments this choice either whenever it can recurse into the previous color (i.e., it chooses a color that is a fragment of the previously selected color), or it can choose a neighbor of the previous color. Among the neighbors, it again maximizes the score as stated above.

Largest Cells. The second cell selector chooses a largest color class, i.e., a color class maximizing $|C|$. The solver contains a version of this cell selector that recurses into fragments of the previous color (similar to the cell selector of Traces, see Section 2.4.4), and one that does not.

Small Cells. The third cell selector chooses a smallest color class, i.e., minimizing $|C|$.

Early Splits. Lastly, there is a cell selector which tries to heuristically find color classes which enable invariant pruning early on in the tree. This is achieved by probing several vertices of a candidate color class. If they lead to differing values of an invariant, they are preferred. (However, this cell selector is only applied if the budget is already very high, such that the cost for probing is amortized by previous restart iterations.)

6.4.2 Restarts

A restart is usually performed whenever the Monte Carlo strategy and breadth-first search exceed the current budget. On each restart, the budget is doubled.

Restart Mitigation. In specific circumstances, the solver does however *not* restart even if the budget is exceeded. These circumstances are designed to model some "common sense" situations in which it seems clear that restarts are detrimental:

1. Equivalent leaves are already being found regularly. In this case, it is likely that the solver is close to completion, and we should not restart anymore.

2. The length of the base is larger than the number of nodes predicted to be computed on the next breadth-first level, and some nodes will be pruned on the next level. In this case, computing the target leaf comes with significant cost, and it seems more beneficial to first do some additional pruning, potentially making the restart and inprocessing more effective.

Partial Base Equivalence. On each restart, the solver checks whether a prefix of the new target leaf agrees with the previous target leaf. If so, this prefix is not discarded in the Schreier structure and breadth-first traversal. If the new target leaf fully agrees with the previous leaf, the solver state of the previous iteration is kept for the next iteration. (If incompatible domain compression is used, the entire Schreier structure is discarded even if prefixes match.)

Evaluating Strategy. In certain circumstances a strategy is immediately discarded, even before performing depth-first search. This is the case in the following circumstances:

1. The estimated IR tree size exceeds the previous tree size.

2. The base is significantly longer than the last base.

We point out that there are further hard limitations in place, to prevent the solver from continuously discarding strategies.

6.5 Inprocessing

On each restart, DEJAVU checks whether the solver state can be used to simplify the graph. In certain circumstances, further invariants are applied.

6.5.1 Simplify using Automorphisms

It is easy to see that whenever we determine an orbit that coincides with the color in the equitable partition, we can individualize an arbitrary vertex of the color.

Lemma 81. *Let (G, π) be a vertex-colored graph and $\langle S \rangle \subseteq \mathrm{Aut}(G, \pi)$. Assume that the orbit partition Δ of $\langle S \rangle$ contains an orbit $\delta \in \Delta$ such that there is a color class C with $C = \delta$. Let π_v denote the coloring in which some $v \in \pi^{-1}(c)$ has been individualized. It follows that $\langle S, \mathrm{Aut}(G, \pi_v) \rangle = \mathrm{Aut}(G, \pi)$.*

Proof. The claim follows immediately from the orbit-stabilizer theorem (Theorem 7). □

We apply the technique for both the automorphisms computed by depth-first search, as well as the random search.

Considering the Schreier structure, we may individualize the base vertex v of level k whenever the orbit matches the color class of the initial vertex coloring of the graph. (Note that each time we individualize a vertex, we modify the initial vertex coloring by individualizing said vertex and then apply color refinement.) Then, we may check for more potential applications of the simplification rules on levels $k' > k$ of the depth-first search. For correctness, observe that automorphisms found at levels $k' > k$ must stabilize the vertex v, i.e., for all automorphisms S found at level k', $\langle S \rangle \subseteq \mathrm{Aut}(G, \pi)_{(v)}$ holds. The same technique can be applied to the computed depth-first levels.

6.5.2 Simplify using Breadth-First Tree

When restarting, we might have computed a partial breadth-first search $T'(G, \pi)$ of the IR tree. Essentially, we want to apply this IR tree as a vertex invariant on the graph G. More precisely, we refine π using the information gathered in the breadth-first tree.

We rephrase aspects of $T'(G, \pi)$ using Lemma 78, such that we may apply them as a node invariant.

Lemma 82. *Let $T'(G, \pi)$ be the IR tree $T(G, \pi)$ up to level k, on which deviation pruning using breadth-first search has been applied.*

Let $v \in V(G)$ and $\varphi \in \mathrm{Aut}(G, \pi)$. The following hold:

1. *If there are k nodes $\mu v \mu' \in T'(G, \pi)$ (where $\mu \in V^c, \mu' \in V^d$), then there are k nodes $\nu v^\varphi \nu' \in T'(G, \pi)$ with $|\mu| = |\nu|$ and $|\mu'| = |\nu'|$.*

2. *If there are k nodes $\mu \in T'(G, \pi)$ where $\pi_\mu(v) = c$ (for some $c \in \{0, \ldots, n-1\}$), then there are k nodes $\nu \in T'(G, \pi)$ such that $\pi_\nu(v^\varphi) = c$ (here, π_μ and π_ν denote the corresponding coloring of the node of the IR tree).*

Proof. The claims follow from the uniformity of the pruning procedure (Lemma 78): if $\mu v \mu' \in T'(G, \pi)$ has not been pruned in the tree, then $\varphi(\mu v \mu') = \nu v^\varphi \nu'$ was not pruned, either. For the second claim, observe that for each $\pi_\mu(v) = c$, $\pi_{\mu^\varphi}(v^\varphi) = c$. □

In the implementation, the above aspects are combined in a hash, which is then used to split color classes of π. This is followed by an application of color refinement.

6.5.3 Simplify using Shallow Search

As discussed previously, the solver will often perform breadth-first search, which is then used as an invariant for the next restart iteration of the solver. In some situations, it is apparent that the solver performs one level of breadth-first search, followed by an immediate restart: this is the case whenever the breadth-first level uses up all the remaining budget of the current restart iteration.

Whenever the solver would perform breadth-first search on the first level and the search would exhaust the remaining budget, the solver chooses to instead immediately restart and apply the invariant described in the following. The idea is that instead of performing a "proper" breadth-first traversal of the level, a "shallow" breadth-first traversal often suffices.

Shallow IR. The idea of the invariant is to only do a *shallow* check for each vertex $v \in V(G)$. The shallow check might distinguish fewer nodes from the target leaf, but is cheaper to compute. The method used is similar to the increased deviation for deviation pruning.

We set a constant c. Then, for each $v \in V(G)$, we individualize v and perform at most c splits in color refinement. The invariant recorded by this limited application of color refinement is stored for vertex v, say, as $\mathrm{Inv}(v)$. Having computed $\mathrm{Inv}(v)$ for

all $v \in V(G)$, we split the initial vertex coloring π according to this invariant. Then, we apply color refinement again to propagate the information further.

In practice, we use the following estimate to choose the constant c: for each computed random walk, we store the point c' at which the walk deviates from the target leaf. For c, we then choose the smallest value observed for c'. Intuitively, this still guarantees us to distinguish *some* of the nodes of the level from the target leaf.

Shallow Multi-Level IR. For some graphs, a single individualization is not sufficient to distinguish any nodes. We extend the method to a heuristic that is able to potentially cover more levels of breadth-first search at once.

Let us fix constants c, d, e. Then, for each $v \in V(G)$, we individualize v and perform at most c splits in color refinement. In the resulting coloring π', we collect all color classes C with $|C| > 1$ and $|C| \leq d$. For each of these color classes, we repeat the process, individualizing all $v \in C$ for each of these color classes C. We do so for at most e levels. (Note that $e = 1$ amounts to the method described above.) We collect a hash of the resulting partial IR tree to split color classes of the initial coloring π, followed by an application of color refinement.

Observe that the shallow IR search as described individualizes vertices indiscriminately. At first, this might seem counter-intuitive: why not use a cell selector and in turn individualize fewer vertices? We believe that, in the way defined above, shallow IR search is *orthogonal* to "restarts and picking a new cell selector". In a sense, restarts amount to a "depth-first" search for a good cell selector, while shallow search is more akin to a "breadth-first" search for cell selectors which lead to distinguishability in the IR tree. Furthermore, using our budget, we are easily able to ensure that the application of the invariant is amortized by computations performed up to that point.

Automorphisms. By definition, for any automorphism $\varphi \in \mathrm{Aut}(G, \pi)$ and any invariant, we know that $v \in V(G)$ and v^φ must receive the same value under the invariant. (This follows by definition of isomorphism-invariance.) Conversely, this means that if a subset $S \subseteq \mathrm{Aut}(G, \pi)$ is already known, we only need to compute the invariant for one vertex of each orbit in $\langle S \rangle$.

We make use of this for our invariant calculations by keeping track of an orbit partition under all applicable automorphisms.

6.6 Parallelization

An earlier version of the solver, described in [5], focused on a parallel implementation. While the implementation as described here is *not* parallelized, we still discuss how the procedures can be parallelized. In particular, we highlight aspects that seem to parallelize well, namely random walks and breadth-first search. We describe which parts need to be synchronized. Lastly, we also discuss the drawbacks of parallelization, and ultimately, why the current version is not parallelized.

6.6.1 Random Search and Sifting

Random walks of the IR tree trivially parallelize due to their independent nature. Once the random walks are computed though, some synchronization is necessary. There are two main aspects:

1. Storing leaves of the tree needs to be synchronized.
2. Storing the Schreier structure and sifting elements needs to be synchronized.

Since the cost of storing and retrieving leaves is negligible, the implementation of [5] simply uses locks to synchronize this task.

Synchronizing the Schreier structure is more involved, and we describe it in the following. Let us make some observations about the Schreier structure, when we are sifting elements for the probabilistic abort criterion.

1. The base is never changed or extended.
2. Changes in the transversal tables T are always local to one level in the Schreier structure.
3. In practice, if sifting is expensive, many elements — probed automorphisms and randomly generated group elements — are sifted. The computationally expensive part is then mostly multiplication of elements (Line 10 of Algorithm 1 and Algorithm 28).

We should stress that in particular, (1) and (2) are generally *not true* when sifting is employed by traditional, deterministic IR algorithms, and are indeed specific to the way it is used by Algorithm 25. This means the algorithm does not need to make base changes, which are generally expensive [104].

Crucially, these three observations enable a rather simple modification to the algorithm: we can sift elements into a shared Schreier structure concurrently, as long as we synchronize local changes to transversal tables when changing a level. Essentially, we only need to add a lock for every level and one global lock for the generating set to enable parallel sifting on a fixed base with sufficient practical performance. These modifications are summarized in Algorithm 28.

We should remark that an even more fine-grained locking mechanism can be easily implemented.

6.6.2 Breadth-First Search

Breadth-first search as used by DEJAVU can be parallelized fairly easily, since computing individual children is independent (unless trace deviation is used, which is discussed further below). For example, we may simply use a queue to share work between threads.

One consideration that has to be made is how color refinement is reversed: if reversible refinement is used (see Section 4.4.3), then preferably all children of a particular node should be computed by the same thread.

Algorithm 28: A thread-safe sifting algorithm.

```
1  function Sift
       Input: ➢ generators S
              ➢ transversal table T
              ➢ base B
              ➢ element φ
      Output: ≺ modifies S and T
              ≺ Boolean whether S and T remained unchanged
2      for ( i = 1; i ≤ |B|; i = i + 1 )
3          b_i := φ(B_i);
4          t := (T_i)_{b_i};
5          if t = ⊥ then break;
6          φ := φ · t^{-1};
7      if φ ≠ id then
8          acquire lock for level i;
9          acquire lock for generators;
10         S := S ∪ {φ};
11         release lock for generators;
12         b_i := φ(B_i);
13         update (T_i)_{b_i} = φ;
14         release lock for level i;
15         return false;
16     return true;
```

Trace Deviation. Making use of trace deviation has a slight synchronization overhead. For trace deviation to be applicable, one first has to compute all children of the target leaf and collect them in a set, before trace deviation pruning can be applied. In order to achieve uniformity of the resulting tree, we need to also ensure that trace deviation pruning *was* applied on all parts of the tree, and siblings are removed. Depending on the implementation, this might require another pass of the resulting breadth-first level.

6.6.3 To parallelize, or not to parallelize?

Parallelization comes with a severe software engineering burden. This is amplified by the circumstance that taming the complexity of the implementation is arguably already an issue for IR algorithms, even without considering parallelization. However, it all boils down to the question: *is it worth it?* Purely from a performance perspective, the answer is certainly *yes* [5]. If IR trees are difficult enough, the parallelization as discussed in this section can lead to a significant speed-up.

However, this misses the fact that in many circumstances, parallelization on the level of symmetry detection is either *not wanted* or *not needed*. For example, in applications such as exhaustive graph generation, parallelization can usually be achieved on a more macroscopic scale. Furthermore, many computational competitions for which symmetry exploitation could be interesting only allow sequential algorithms. If symmetry detection is to be used as a subroutine, an implementation that only works well in parallel is therefore undesirable.

In particular, the previous parallel implementation described in [5] used strategies that did not work well sequentially, essentially trading sequential performance for parallel performance. On the other hand, the sequential implementation described in this thesis features routines which do not trivially parallelize (e.g., the preprocessor, depth-first search phase, restart scheme). Observe that these include precisely the routines which are heavily used for large, practical graphs. However, using a more condensed approach to parallelization, i.e., by only parallelizing breadth-first search and random search for large budgets, one should be able to recover most of the benefits of the parallel implementation, while maintaining the sequential efficiency of the current implementation.

Chapter 7

Benchmarks

For *practical* algorithms, it is of course paramount to test the effectiveness of algorithms *in practice*. We compare the C++ implementation of DEJAVU to all state-of-the-art tools, namely NAUTY, SAUCY, BLISS, and TRACES. Furthermore, we gauge the effectiveness of the preprocessor for NAUTY, SAUCY, BLISS, and TRACES.

Let us begin with some technicalities. All benchmarks ran on an Intel i7 9700K with 64GB of RAM on Ubuntu 20.04. The solvers used a single thread with the full amount of RAM available (we did not run any benchmarks in parallel). The time limit is 100 seconds. If a solver runs out of memory or crashes for any other reason, this counts as a time out (only NAUTY and TRACES occasionally ran out of memory). All graphs were randomly permuted, but each solver was passed the same permuted version of each graph. The solver versions used are BLISS 0.73, NAUTY/TRACES 2.6R12, SAUCY 3.0, and DEJAVU 2.0. The source code of DEJAVU is available here [31]. (In particular, the precise version tested in this thesis is archived [31].)

We configured DEJAVU with an error probability below 0.1% (using a pseudo random number generator). Essentially, we used the default configuration of the solver. However, we did make one adjustment for the sake of fairness: we artificially ensured that all generators are lifted back to the original graph (see Chapter 5). If only the automorphism group size is computed, DEJAVU will not do so, in turn potentially running faster.

Regarding the probabilistic one-sided error, we checked the reported automorphism group sizes of DEJAVU against the automorphism group sizes reported by a deterministic solver whenever this was possible – i.e., whenever there was a deterministic solver which did not time out. In our benchmarks, no probabilistic error for DEJAVU could be observed: the automorphism group sizes reported by the solvers always agreed.

7.1 Graph Library

To compare solvers in a meaningful way, it is crucial to test them on a well-rounded benchmark suite that covers a wide variety of interesting graphs. We list all the graph classes tested in our benchmarks. Most of the graphs are from the collection of NAUTY/TRACES, obtained from [85]. The collection combines most graph classes tested in prior publications regarding practical graph isomorphism solvers. We add further graph classes to the suite, most of which are directly motivated by practical applications. The additional classes are described in Section 7.1.2.

7.1.1 Graph Classes from the nauty/Traces Collection

We give a brief description of the graph classes used from the NAUTY/TRACES collection [85]. Note that we include almost all graph classes listed in [85]. We only excluded the suite of Brendan McKay, which is mostly comprised of very small graphs. Furthermore, we exlcuded some variants of the multipedes [86]. Testing these classes takes a substantial amount of time, and would most likely be largely redundant to testing the shrunken multipedes (see [86]).

We recall the descriptions as given in [58, 28, 70, 85, 86]. The classes ag [58] and pg [58] contain bipartite point-line incidence graphs of 2-dimensional affine and projective geometries. The class pp [58] consists of bipartite point-line incidence graphs of projective planes. The class cfixl (originally from [58], but extended with larger graphs) consists of random 3-regular graphs augmented using the construction of [20]. The class had [58] consists of graphs built from Hadamard matrices, whereas the graphs in had-sw are augmented further using switching operations to reduce symmetry [88]. The class latin [58] contains graphs stemming from latin squares, whereas latin-sw again contains graphs obtained from symmetry-reducing switching operations. Similarly, sts [58] stem from Steiner triple systems, and sts-sw are said graphs with reduced symmetry. Furthermore, there are the Kronecker eye flip graphs kef [85].

The Miyazaki graph classes cmz, mz, mz-aug, and mz-aug2 [58] are based on the construction of [82], describing adversarial graphs for IR algorithms, aimed at particular cell selectors. The shrunken multipedes multipedes are asymmetric benchmark graphs designed to cause IR trees of exponential size [86]. The Dawar-Yeung graphs dy are based on random 3-XOR formulas, and were also designed with the aim of providing hard instances for IR algorithms [85, 29].

The class dac [85] contains model graphs for CNF SAT formulas, internet represents interconnections of major routers on the internet [28], ispd is derived from circuits of the ISPD 2005 placement competition [28], and states [85] contains the road networks of US states.

The class k consists of complete graphs, grid [58] of grid graphs, and grid-w[58] of grid graphs with a wrapped boundary. The class ranreg [85] consists of random 6-regular graphs, whereas rnd-3-reg[58] contains random 3-regular graphs. The class tran [85] is a collection vertex transitive graphs, combinatorial a collection of combinatorial graphs by Gordon Royle [85], and f-lex a collection of product graphs by Petteri Kaski [85].

The class chh contains tailored graphs built from hypo-Hamiltonian graphs [70]. Graphs of the class tnn are built from two tripartite graphs [70]. The class usr contains unions of strongly regular graphs [70].

The classes ran2, ran10, ransq [85] contain Erdős-Rényi graphs with various edge probabilities. The class rantree [85] contains random trees. The class hypercubes contains, unsurprisingly, hypercubes.

7.1.2 Additional Graph Classes

We extend the suite above with the following graph classes.

The first graph class sat21 contains model graphs for instances of the SAT competition 2021 [102]. As discussed in Chapter 1, the use of symmetry in SAT is of great interest and hence it seems only natural to test a collection of contemporary instances. Similarly, the second class mip17 contains model graphs stemming from a MIP benchmark library [43, 81]. The third class is pace23, and is simply comprised of all the graphs used in the PACE challenge 2023 [90]. Lastly, groups128 contains graphs obtained from the Caley tables of several groups of order 128, sampled from the small groups library of GAP [41].

7.2 dejavu versus State-of-the-Art

Let us first consider the results of our benchmarks separated into graph classes. The results for each graph class are summarized in Table 7.1. First, taking the sum of all the runtimes of a graph class, DEJAVU is fastest on 33 classes, with TRACES being fastest on 7, SAUCY on 3, and NAUTY on 1 (out of 44 total classes). We define a solver to be *competitive*, if it finishes within a factor of two of the fastest solver. Considering this, DEJAVU is competitive on 39 classes, TRACES on 22, SAUCY on 15, NAUTY on 10, and BLISS on 3 (out of 44 total classes). On all graph classes, DEJAVU posted the lowest number of timeouts achieved in the benchmarks (in the majority of classes the score was tied at 0, though). Summing over all the graphs in the benchmark suite, DEJAVU reported 27 timeouts, TRACES reported 558, BLISS reported 489, NAUTY reported 840, and SAUCY reported 557 (out of 4259 graphs). We also remark that NAUTY and TRACES ran out of memory on some graphs, which we counted as a timeout.

Figure 7.1 illustrates the results on all the tested graphs individually. Each diagram shows the results of one state-of-the-art solver compared to the results of DEJAVU. Here, SAUCY is faster than DEJAVU on 34.9% of all graphs, TRACES is faster on 25.3%, NAUTY is faster on 15.6%, and BLISS is faster on 12.5%. Hence, in our benchmarks, compared to any other solver, DEJAVU is faster on the majority of individual graphs. A trend that is quite apparent in these figures is that DEJAVU tends to be comparatively slower on graphs that are solved very quickly. The solver starts to gain an advantage as graphs become more challenging (i.e., as they become more difficult or larger). However, in the plots, we can also readily observe that graphs on which DEJAVU wins tend to be further away from the diagonal: on average, DEJAVU wins by a larger margin. This is exemplified by the fact that there is only one graph on which DEJAVU times out, but there is a solver which does not. (TRACES solves one particular instance of the sat21 graph class within the timeout, which DEJAVU does not.)

These observations match expectation: the high-level algorithm of DEJAVU probes for good strategies, and potentially employs several techniques before solving a graph. It is to be expected that this overall process comes with a certain overhead. This in turn sometimes leads to diminished performance on easy, and typically very small, graphs. However, on the other hand, DEJAVU has many more options for finding a good way of solving a graph when needed, which leads to the improved performance on more involved graphs.

Going into more detail, Figure 7.2 up to Figure 7.45 plot the results for each graph class

set		DEJAVU		TRACES		BLISS		NAUTY		SAUCY	
name	size	t	t/o	t	t/o	t	t/o	t	to	t	t/o
ag	23	*0.097*	**0**	**0.084**	**0**	0.29	**0**	0.20	**0**	1804	18
cfixl	101	**51**	**0**	*95*	**0**	806	7	765	7	1000	10
chh	26	**0.14**	**0**	477	4	286	2	1217	11	1238	12
cmz	46	*0.031*	**0**	0.18	**0**	74	**0**	3418	33	**0.024**	**0**
combinatorial	12	**21**	**0**	63	**0**	527	5	847	7	1060	10
dac	29	**0.40**	**0**	2.1	**0**	1.8	**0**	31	**0**	64	**0**
dy	290	**231**	**0**	7477	72	*261*	1	*321*	**0**	7198	64
f-lex	210	**34**	**0**	16619	162	8733	83	19457	192	395	**0**
grid	39	*0.071*	**0**	**0.044**	**0**	0.096	**0**	*0.062*	**0**	*0.070*	**0**
grid-w	39	*0.12*	**0**	*0.078*	**0**	0.17	**0**	**0.066**	**0**	*0.10*	**0**
groups128	17	**12**	**0**	37	**0**	43	**0**	342	2	333	1
had	66	**11**	**0**	*21*	**0**	48	**0**	470	3	1254	7
had-sw	51	8.5	**0**	**3.0**	**0**	202	1	523	3	807	4
hypercubes	19	**93**	**0**	*100*	**0**	*186*	1	*155*	1	230	1
internet	3	**0.16**	**0**	2.4	**0**	198	1	300	3	*0.19*	**0**
ispd	8	**5.7**	**0**	*6.4*	**0**	800	8	800	8	*7.4*	**0**
k	112	**0.053**	**0**	41	**0**	370	3	122	**0**	0.30	**0**
kef	11	*0.005*	**0**	0.056	**0**	0.029	**0**	0.065	**0**	**0.005**	**0**
latin	29	0.077	**0**	**0.031**	**0**	0.17	**0**	*0.058*	**0**	*0.040*	**0**
latin-sw	231	**3.6**	**0**	*5.0*	**0**	44	**0**	444	**0**	281	**0**
lattice	27	**0.024**	**0**	0.065	**0**	0.33	**0**	0.057	**0**	0.13	**0**
mip17	240	**14**	**0**	1103	9	174	**0**	1470	10	*22*	**0**
multipedes	348	**4336**	**25**	21452	214	12107	114	12019	108	15170	145
mz	25	0.067	**0**	**0.013**	**0**	0.14	**0**	0.15	**0**	1003	10
mz-aug	25	0.11	**0**	**0.015**	**0**	0.13	**0**	0.090	**0**	1403	14
mz-aug2	24	0.010	**0**	*0.005*	**0**	0.19	**0**	1739	17	**0.005**	**0**
pace23	400	**35**	**0**	5736	51	3927	26	10149	90	133	**0**
paley	53	**0.018**	**0**	*0.020*	**0**	0.32	**0**	*0.020*	**0**	0.046	**0**
pg	23	**0.083**	**0**	*0.087*	**0**	0.36	**0**	0.27	**0**	*0.11*	**0**
pp	243	**606**	**0**	2661	1	21366	203	20702	196	22800	228
ran2	131	**0.077**	**0**	0.19	**0**	0.55	**0**	*0.13*	**0**	0.28	**0**
ran10	150	**0.12**	**0**	0.27	**0**	0.86	**0**	*0.21*	**0**	0.42	**0**
ransq	150	**0.033**	**0**	*0.054*	**0**	0.14	**0**	*0.057*	**0**	*0.062*	**0**
rantree	19	*0.033*	**0**	**0.022**	**0**	2.4	**0**	230	2	*0.026*	**0**
ranreg	13	**0.40**	**0**	4.7	**0**	155	1	437	4	4.9	**0**
rnd-3-reg	110	**0.73**	**0**	*1.1*	**0**	28	**0**	2081	**0**	3.0	**0**
sat21	400	**1531**	**2**	4231	28	4242	26	11057	88	*2387*	12
states	56	**7.1**	**0**	*12*	**0**	1538	7	5120	51	*7.6*	**0**
sts	25	**0.21**	**0**	*0.36*	**0**	0.64	**0**	14	**0**	9.5	**0**
sts-sw	231	**3.1**	**0**	*5.4*	**0**	33	**0**	659	**0**	396	**0**
tnn	20	**0.33**	**0**	403	4	1.7	**0**	7.4	**0**	905	9
tran	139	**1.1**	**0**	4.9	**0**	6.6	**0**	7.7	**0**	*1.6*	**0**
triang	27	**0.012**	**0**	*0.018*	**0**	0.13	**0**	*0.021*	**0**	0.046	**0**
usr	18	**4.6**	**0**	1306	13	*8.8*	**0**	441	4	1203	12

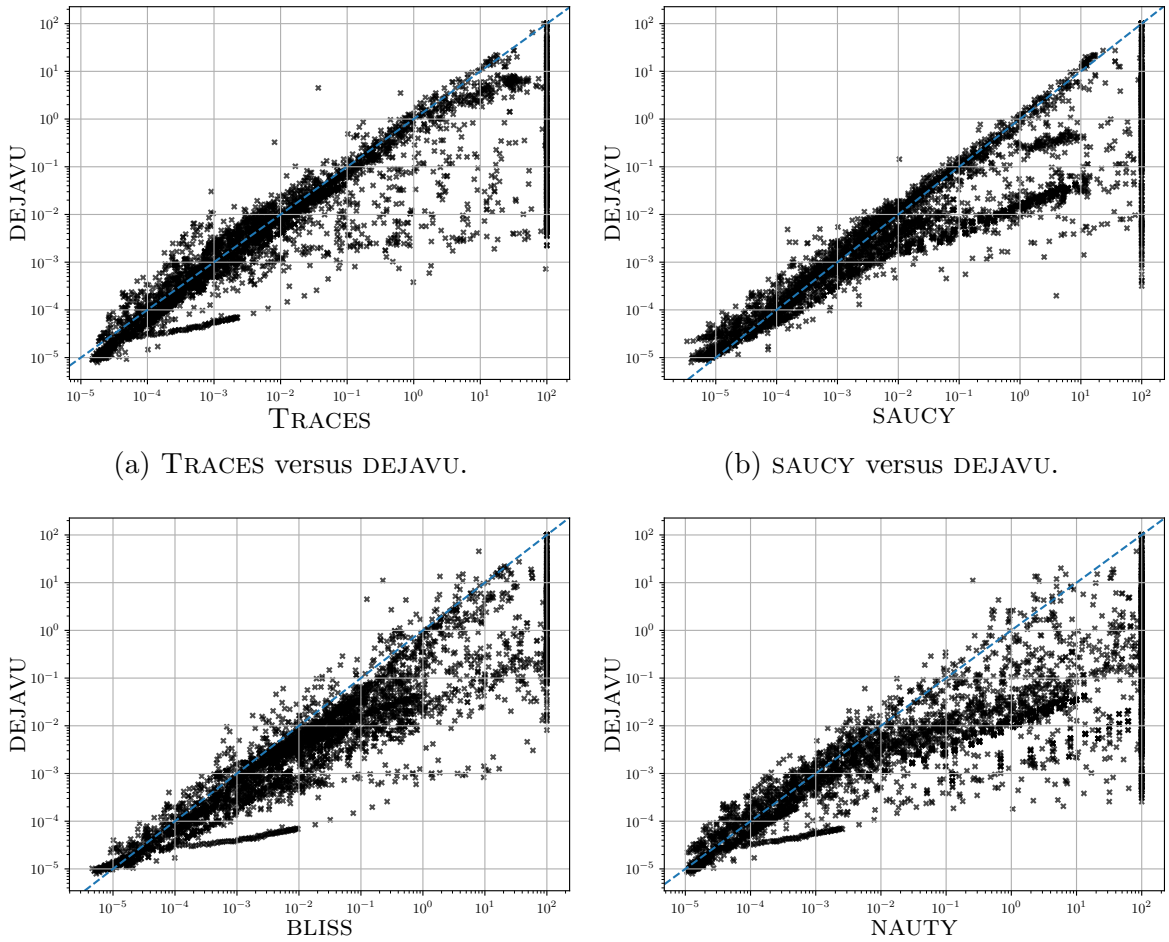
Table 7.1: Benchmark results for each graph class. Columns labeled "t" contain the sum of the runtime in seconds for each graph class. Columns labeled "t/o" record the number of timeouts (100 seconds) on the graph class.

individually. The (a) part of each figure orders the results for each solver by runtime, whereas (b) distributes the instances according to their number of vertices. Let us remark on some notable observations.

On the shrunken multipedes (Figure 7.24), which are essentially worst-case examples for IR algorithms, benchmarks match the better asymptotic scaling we would expect of the Monte Carlo algorithm.

For random regular graphs (Figure 7.36 and Figure 7.37) and f-lex (Figure 7.9), DEJAVU heavily relies on the shallow IR inprocessing technique (see Section 6.5). The data suggest that the technique exhibits an asymptotic advantage over the search strategies of the other solvers.

On some of the Miyazaki graph classes (Figure 7.5, Figure 7.25, Figure 7.26 and Figure 7.27), DEJAVU is not competitive to the fastest solver. This seems to be due to inconsistent and unnecessary restarts, which lead to a higher overall runtime, but not necessarily worse scaling. The same seems to be true for Latin squares (Figure 7.20): in particular, we observe that different permutations of a graph can lead to significantly different numbers of restarts, which in turn sometimes cause unnecessary overhead, and other times it does not.



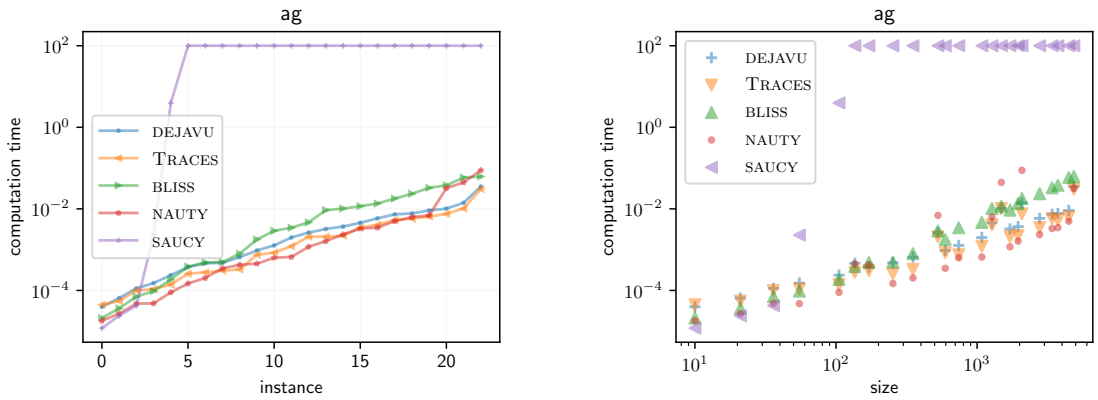
(a) TRACES versus DEJAVU.

(b) SAUCY versus DEJAVU.

(c) BLISS versus DEJAVU.

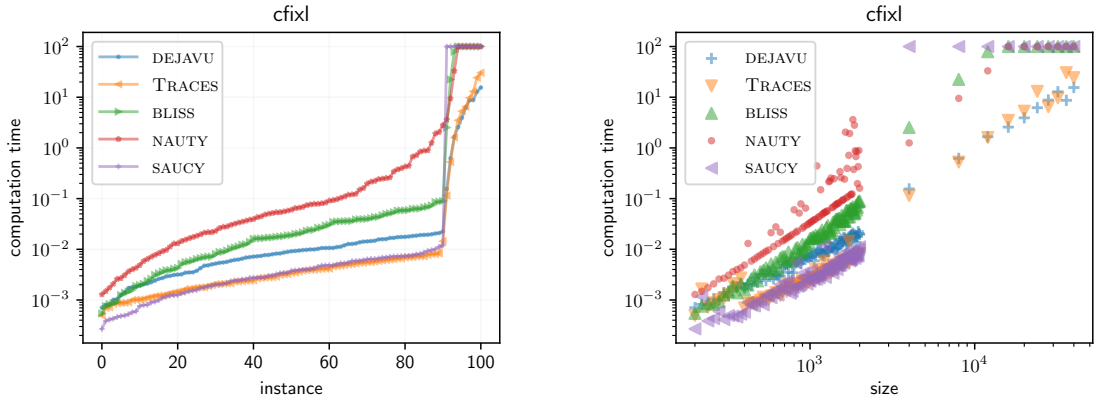
(d) NAUTY versus DEJAVU.

Figure 7.1: Comparing state-of-the-art solvers on all tested graphs. The y-axis is the runtime of DEJAVU in seconds, and the x-axis is the runtime of the other solver. Points below the diagonal indicate DEJAVU being faster.



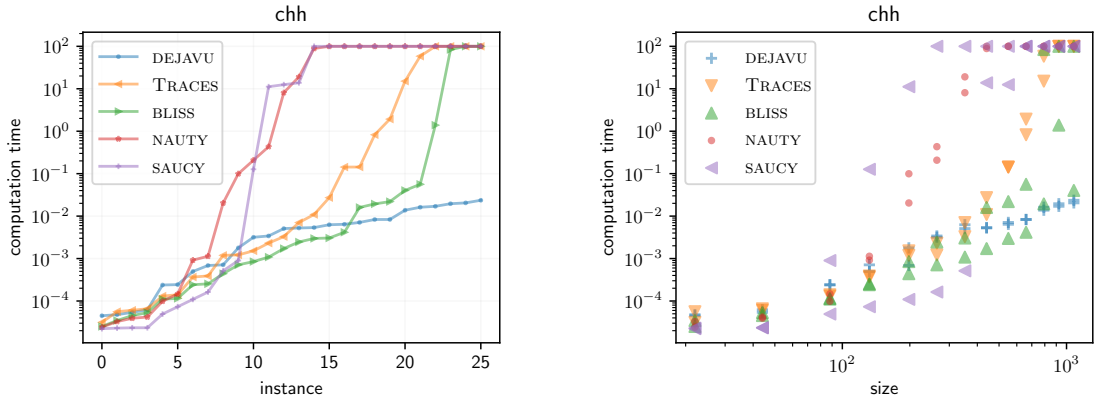
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.2: Benchmark results for the graph class `ag`.



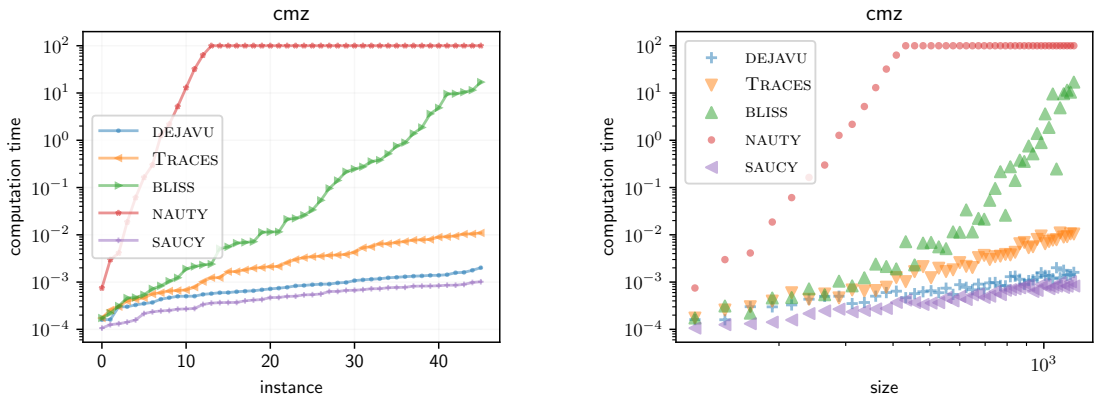
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.3: Benchmark results for the graph class `cfixl`.



(a) Sorted according to runtime. (b) Sorted according to size.

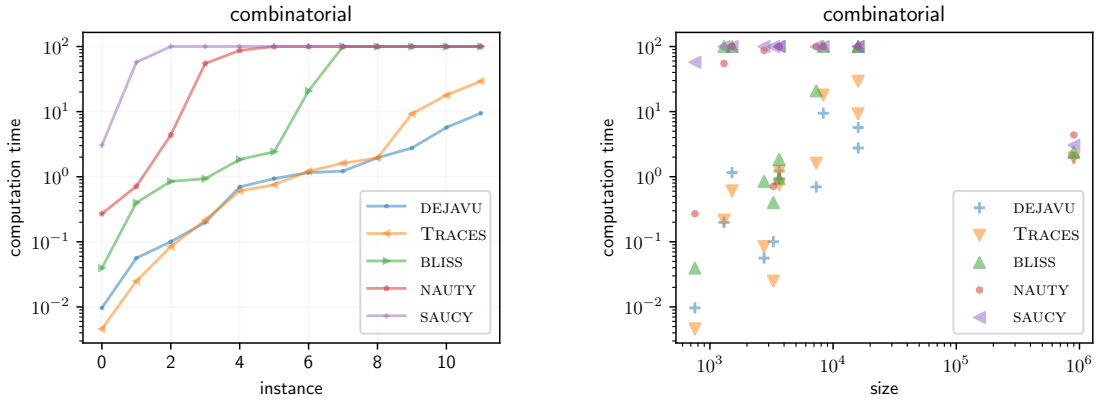
Figure 7.4: Benchmark results for the graph class `chh`.



(a) Sorted according to runtime.

(b) Sorted according to size.

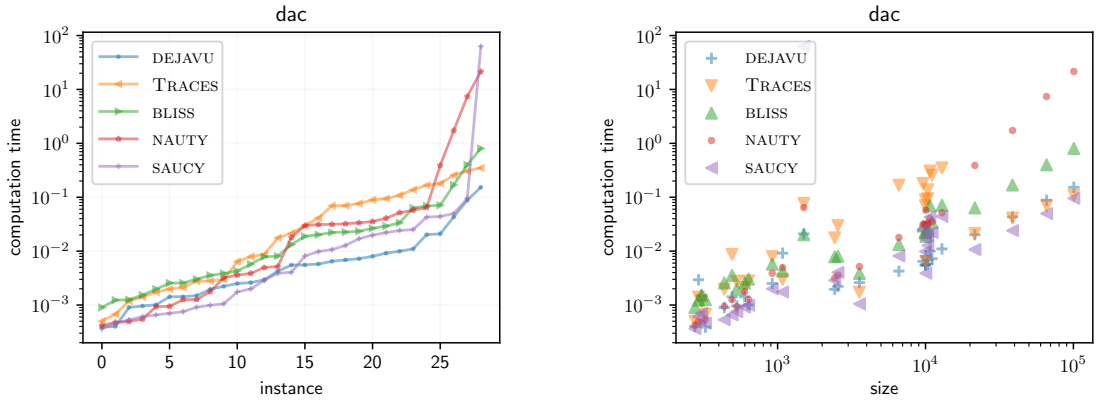
Figure 7.5: Benchmark results for the graph class `cmz`.



(a) Sorted according to runtime.

(b) Sorted according to size.

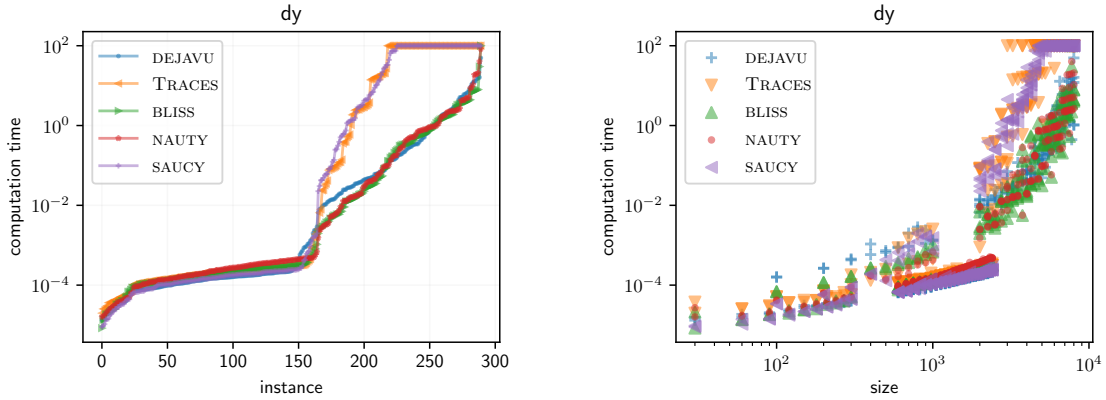
Figure 7.6: Benchmark results for the graph class `combinatorial`.



(a) Sorted according to runtime.

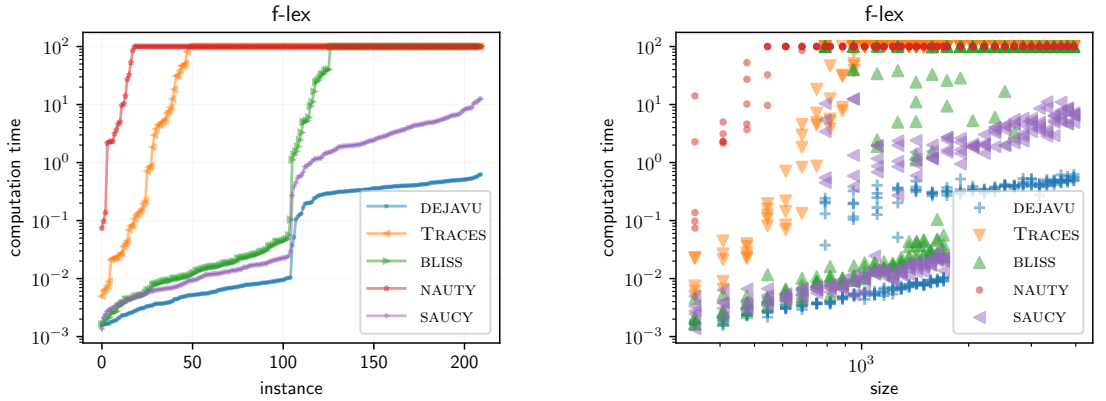
(b) Sorted according to size.

Figure 7.7: Benchmark results for the graph class `dac`.



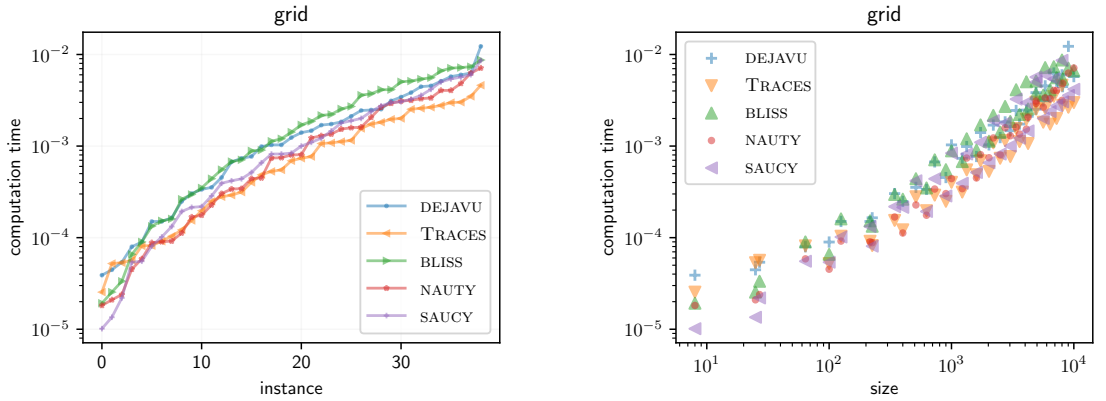
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.8: Benchmark results for the graph class `dy`.



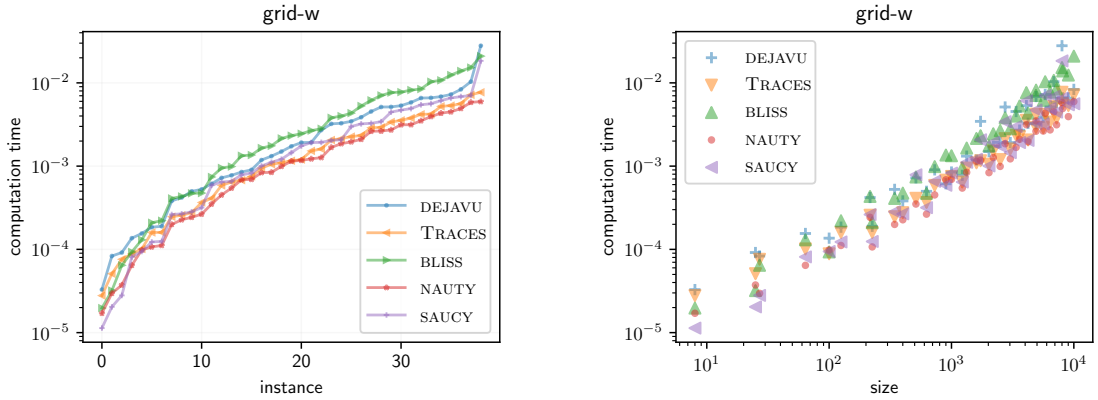
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.9: Benchmark results for the graph class `f-lex`.



(a) Sorted according to runtime. (b) Sorted according to size.

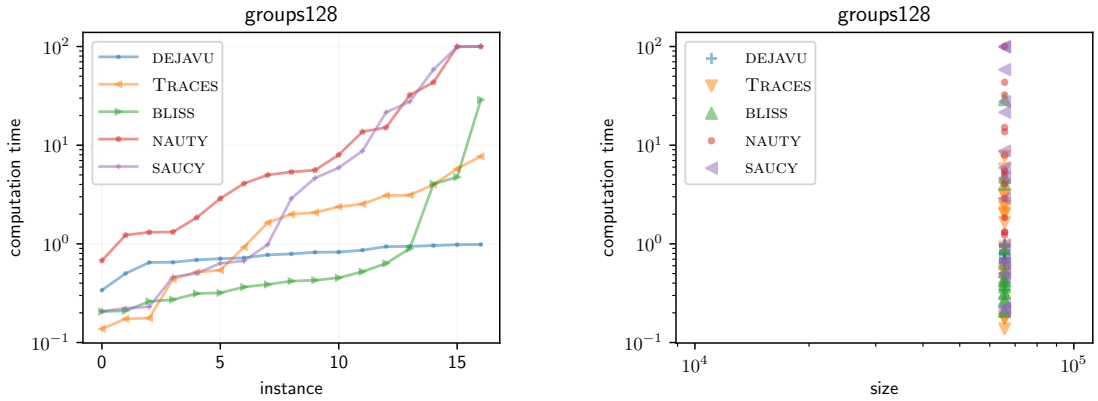
Figure 7.10: Benchmark results for the graph class `grid`.



(a) Sorted according to runtime.

(b) Sorted according to size.

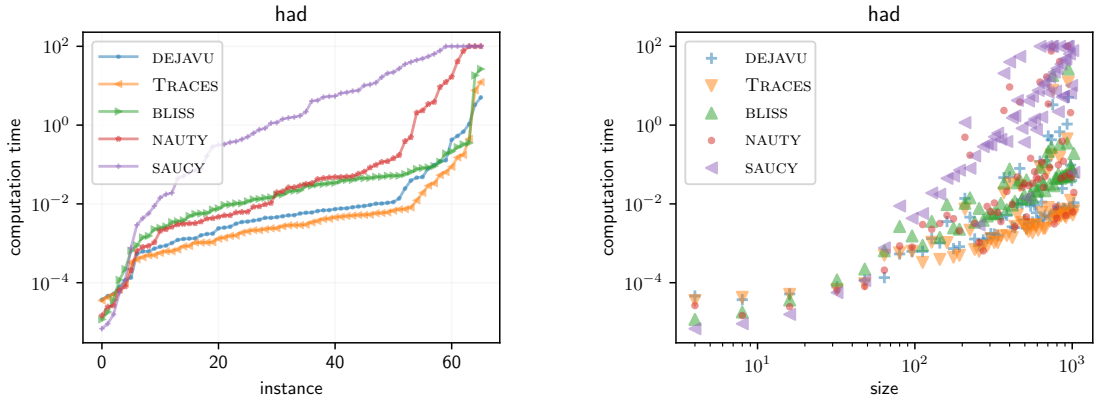
Figure 7.11: Benchmark results for the graph class `grid-w`.



(a) Sorted according to runtime.

(b) Sorted according to size.

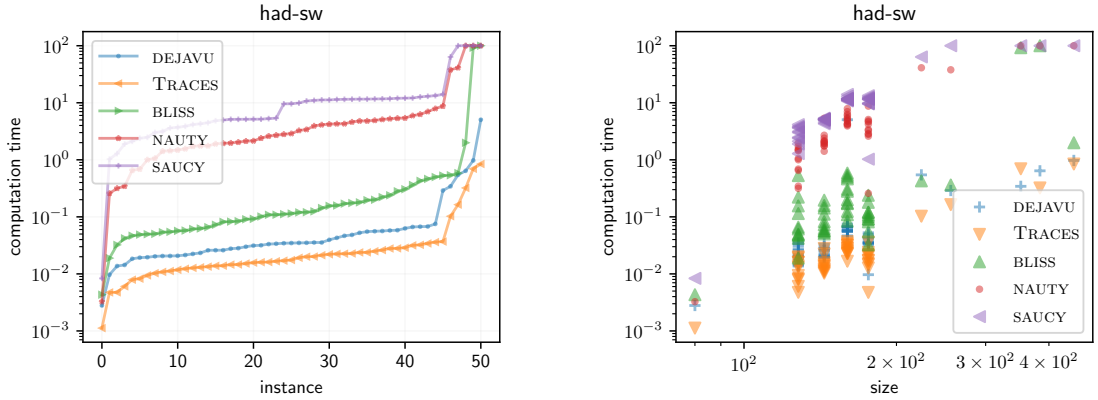
Figure 7.12: Benchmark results for the graph class `groups128`.



(a) Sorted according to runtime.

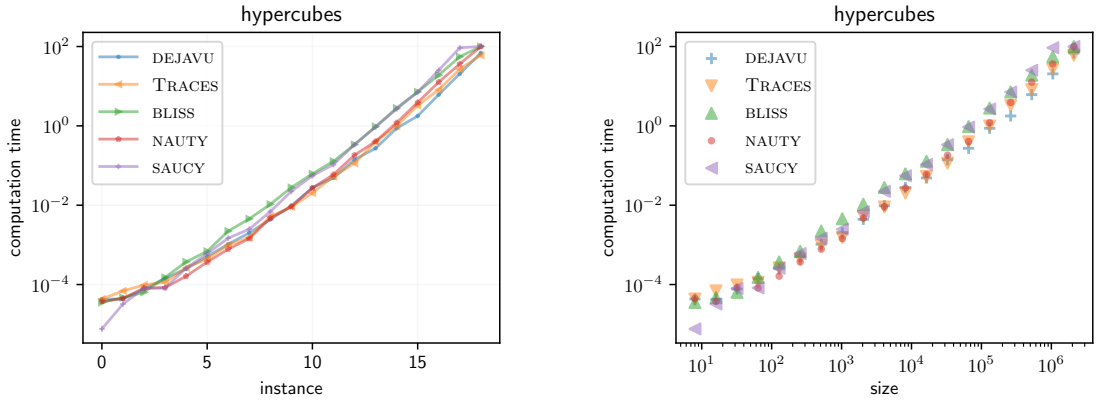
(b) Sorted according to size.

Figure 7.13: Benchmark results for the graph class `had`.



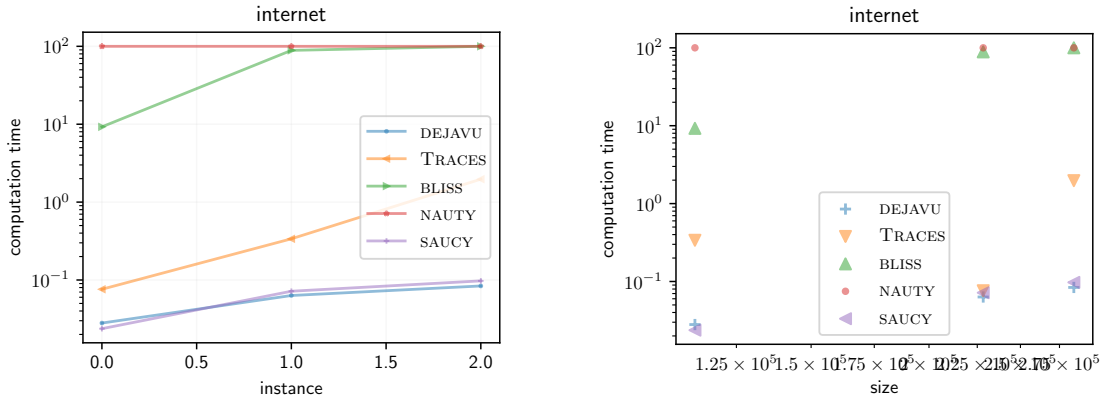
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.14: Benchmark results for the graph class `had-sw`.



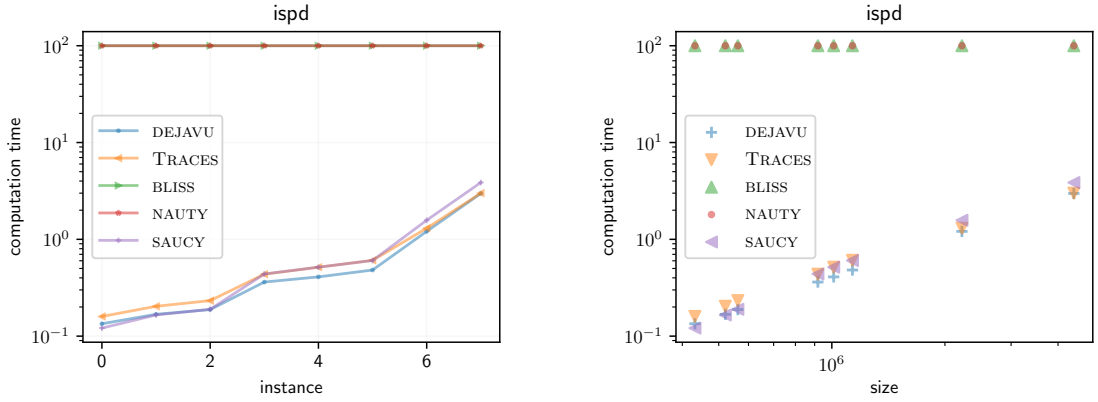
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.15: Benchmark results for the graph class `hypercubes`.



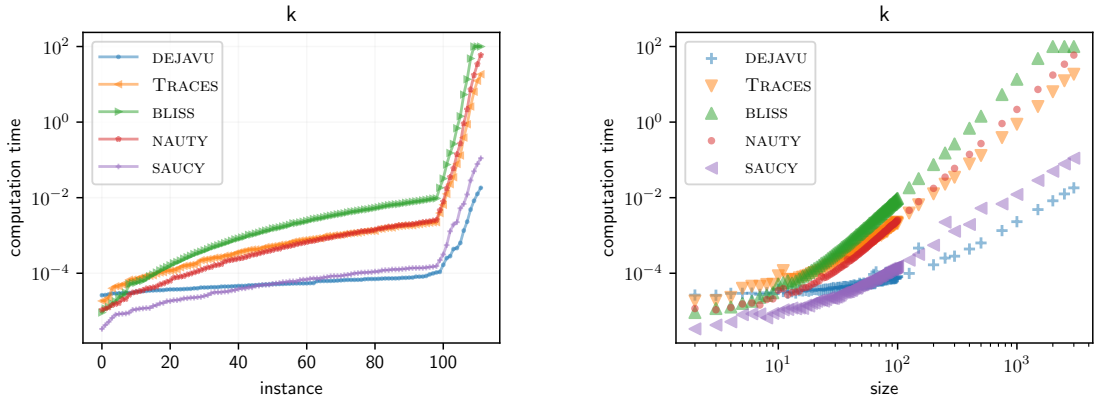
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.16: Benchmark results for the graph class `internet`.



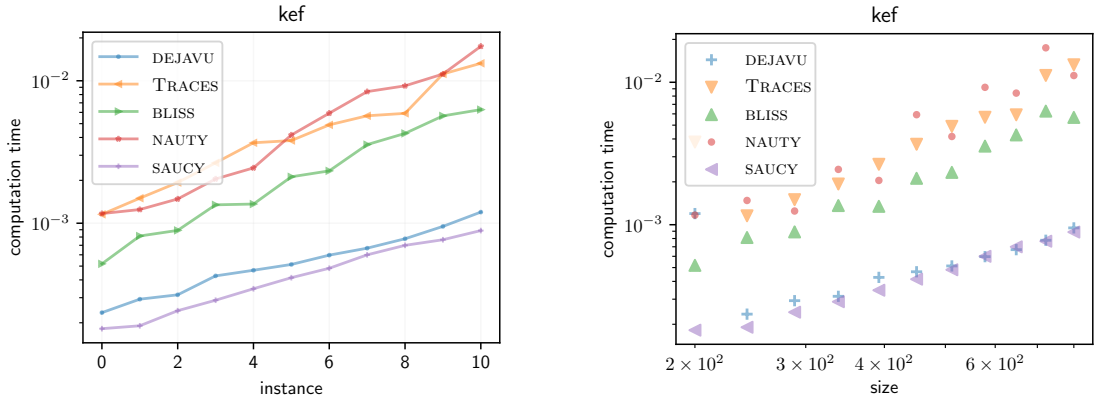
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.17: Benchmark results for the graph class `ispd`.



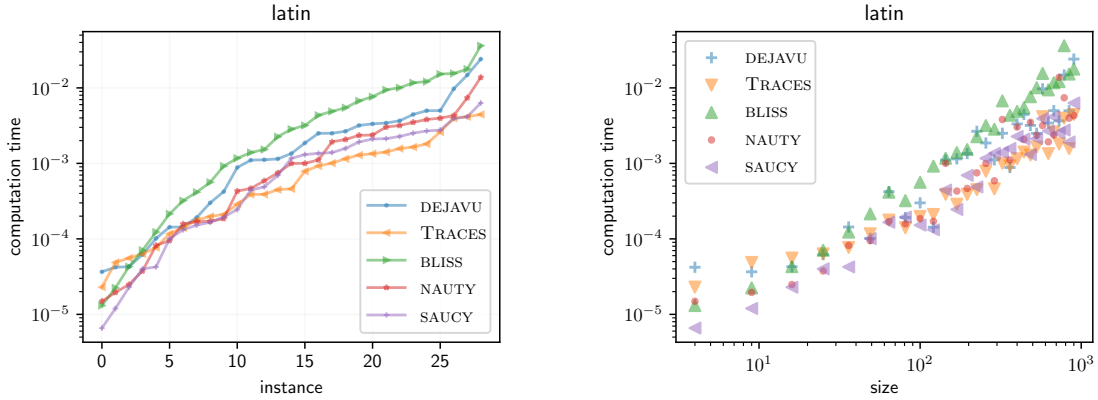
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.18: Benchmark results for the graph class `k`.



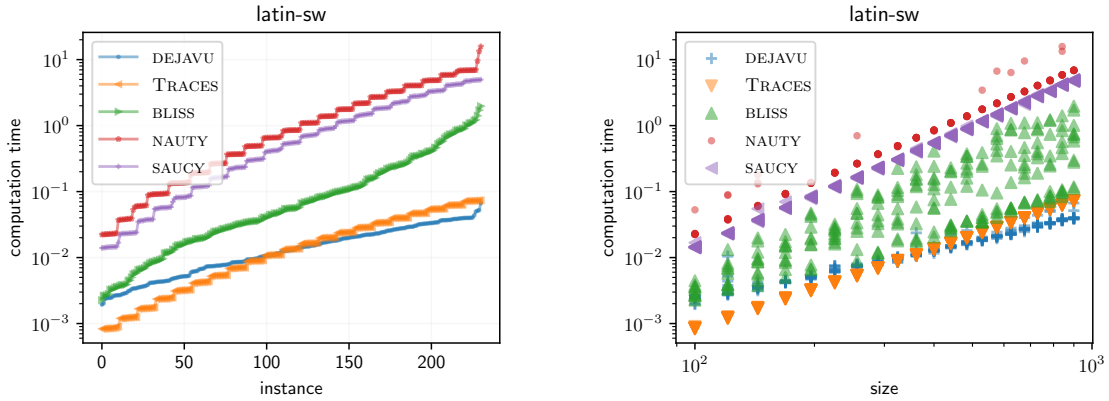
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.19: Benchmark results for the graph class `kef`.



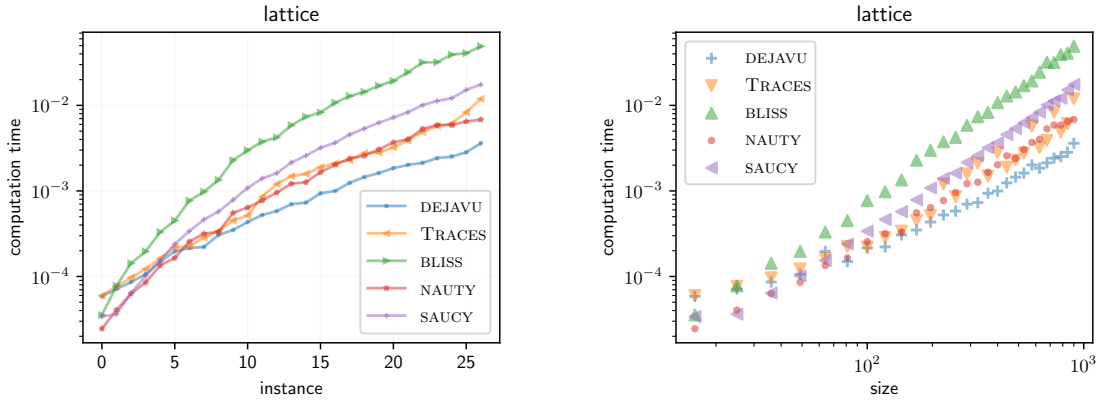
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.20: Benchmark results for the graph class `latin`.



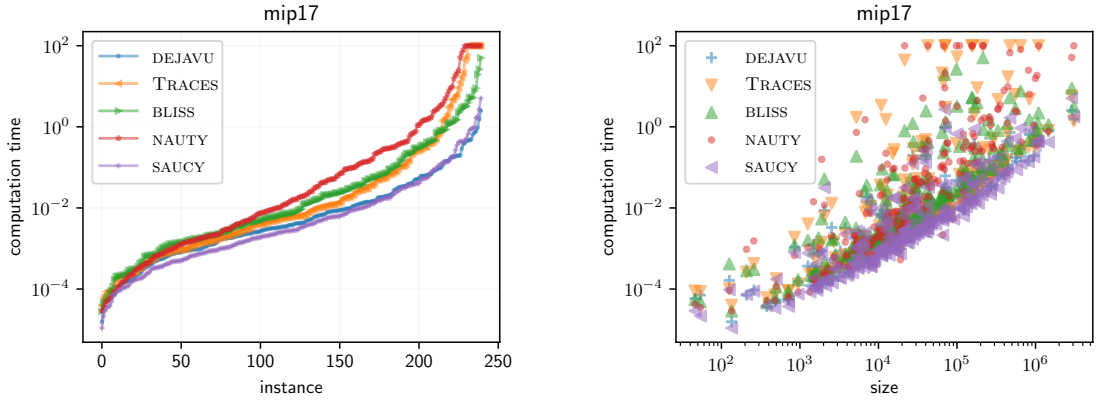
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.21: Benchmark results for the graph class `latin-sw`.



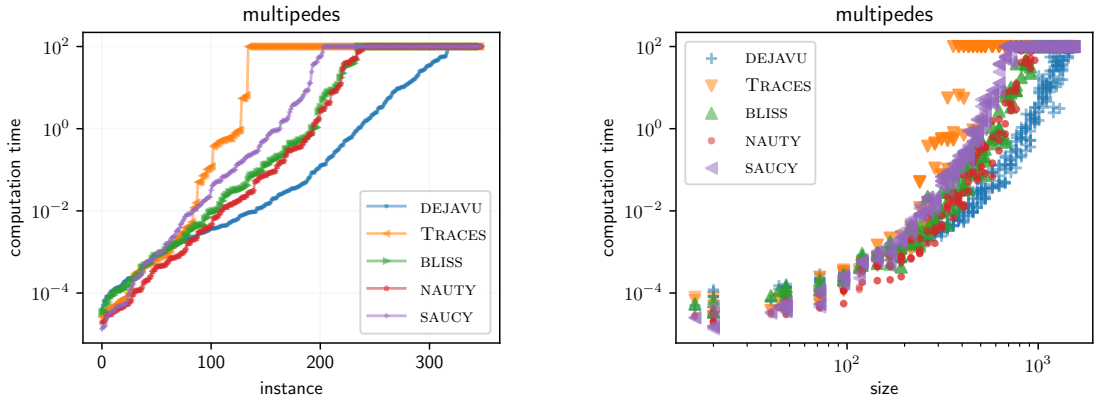
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.22: Benchmark results for the graph class `lattice`.



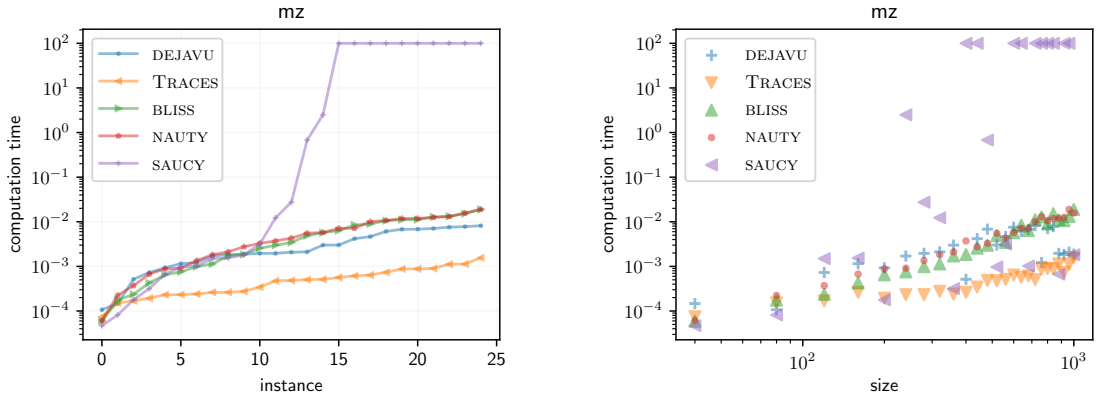
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.23: Benchmark results for the graph class `mip17`.



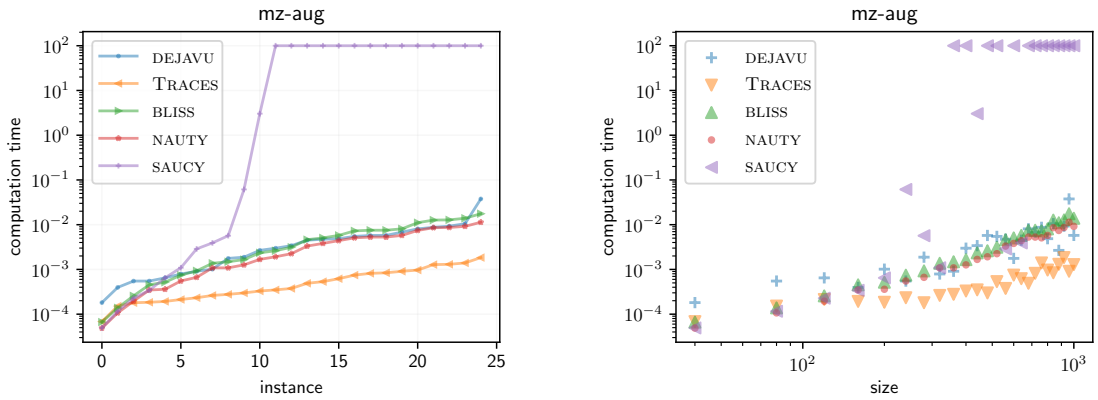
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.24: Benchmark results for the graph class `multipedes`.



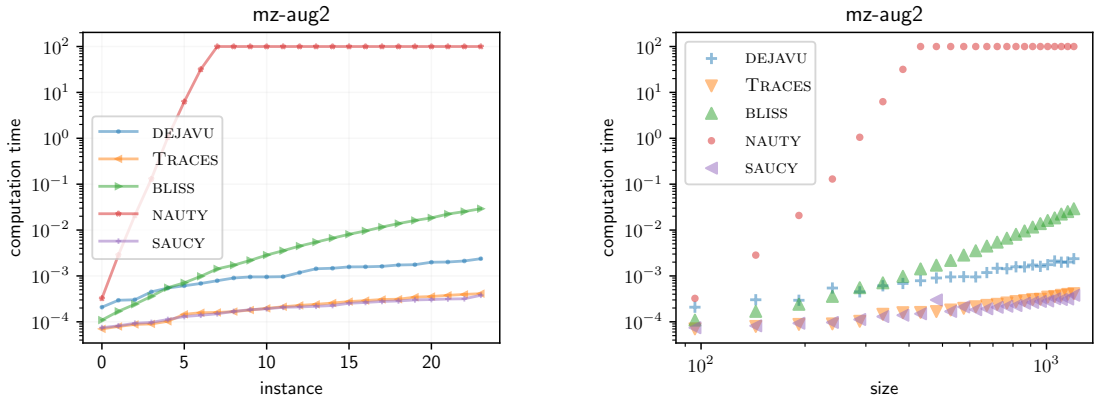
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.25: Benchmark results for the graph class `mz`.



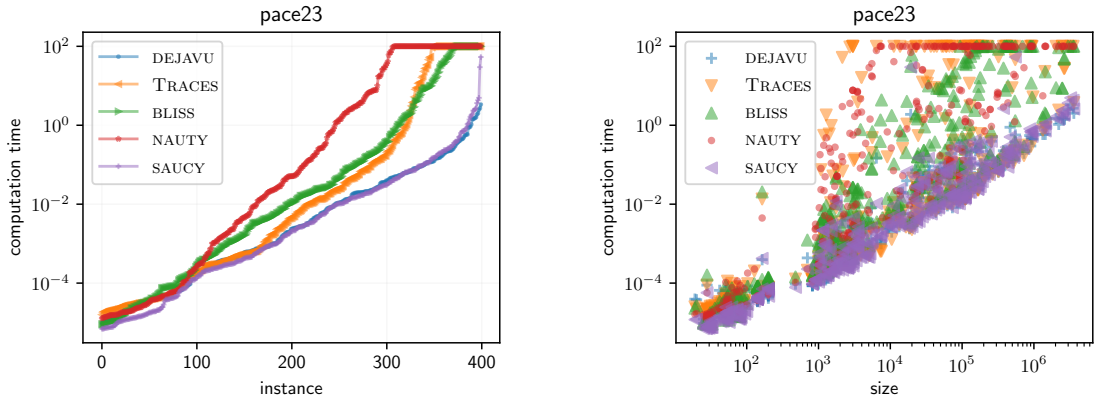
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.26: Benchmark results for the graph class `mz-aug`.



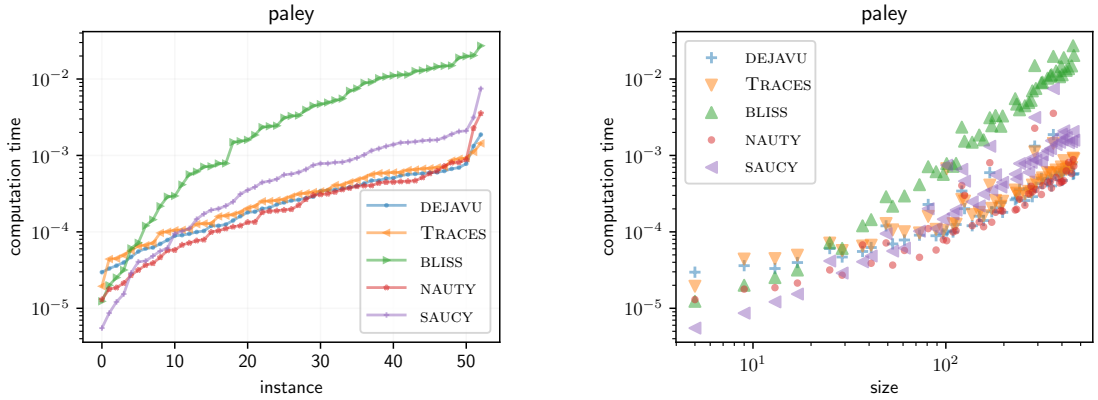
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.27: Benchmark results for the graph class `mz-aug2`.



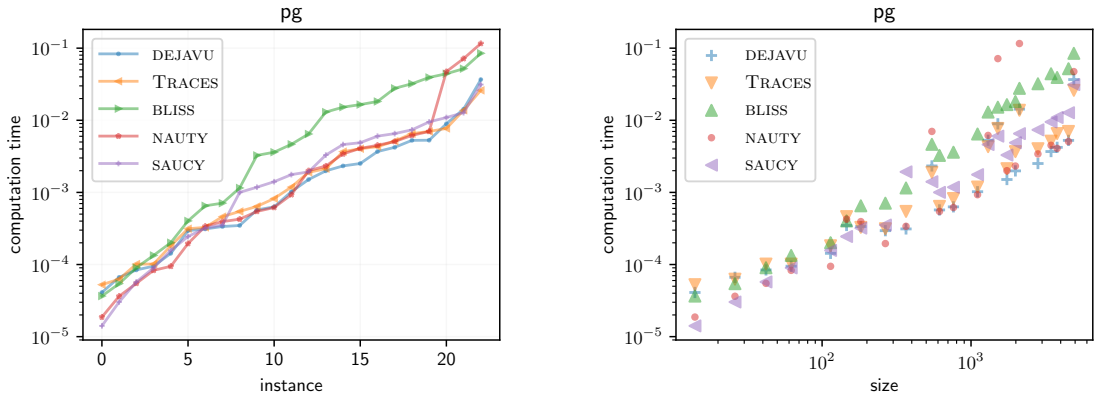
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.28: Benchmark results for the graph class `pace23`.



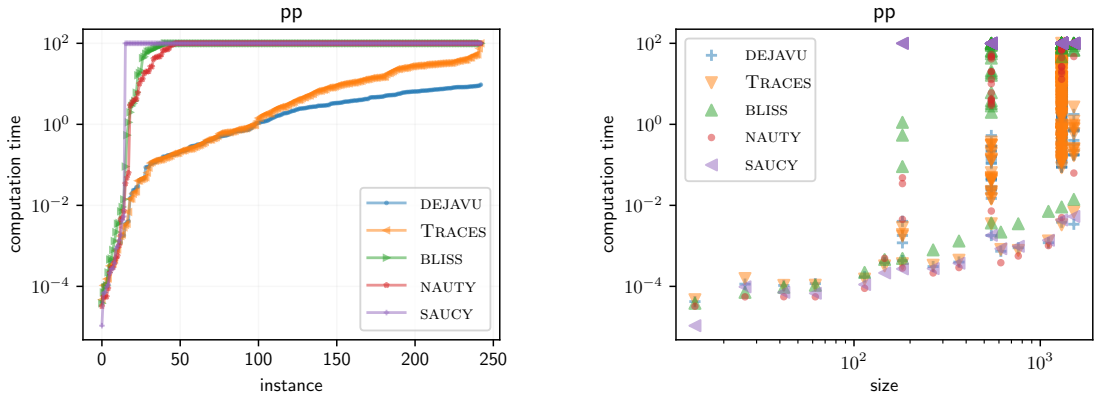
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.29: Benchmark results for the graph class `paley`.



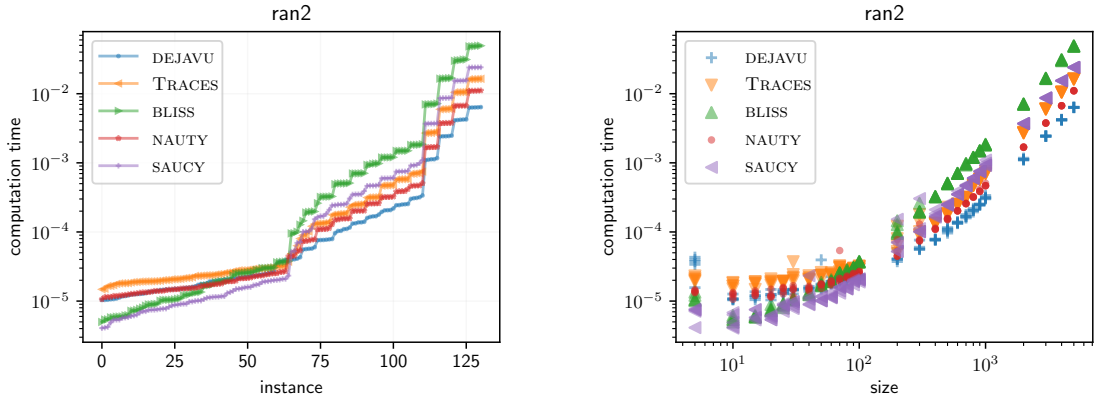
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.30: Benchmark results for the graph class `pg`.



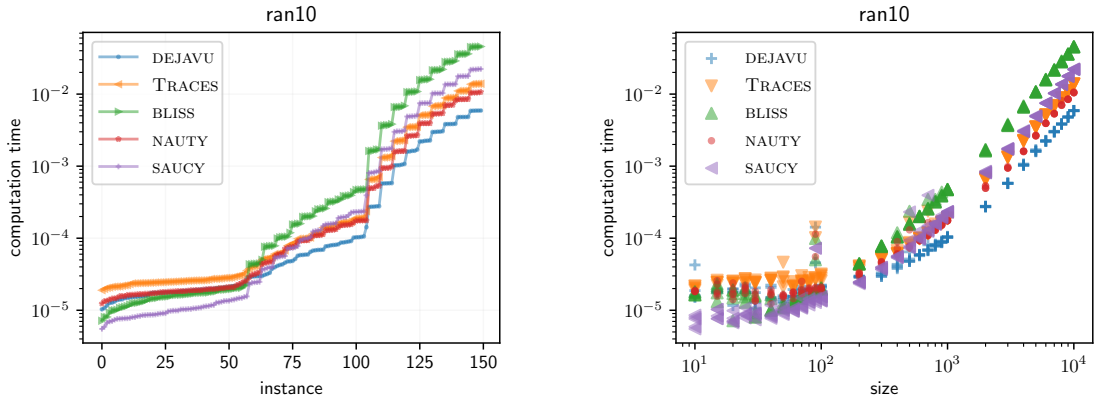
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.31: Benchmark results for the graph class `pp`.



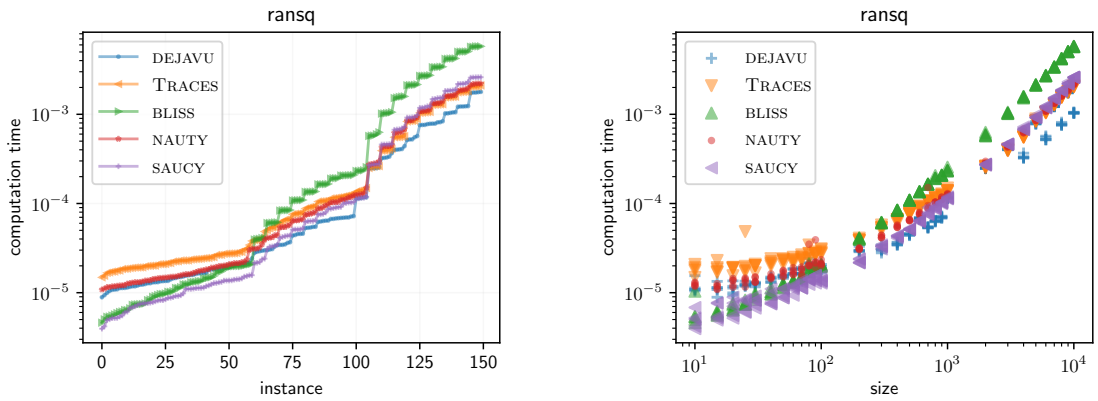
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.32: Benchmark results for the graph class `ran2`.



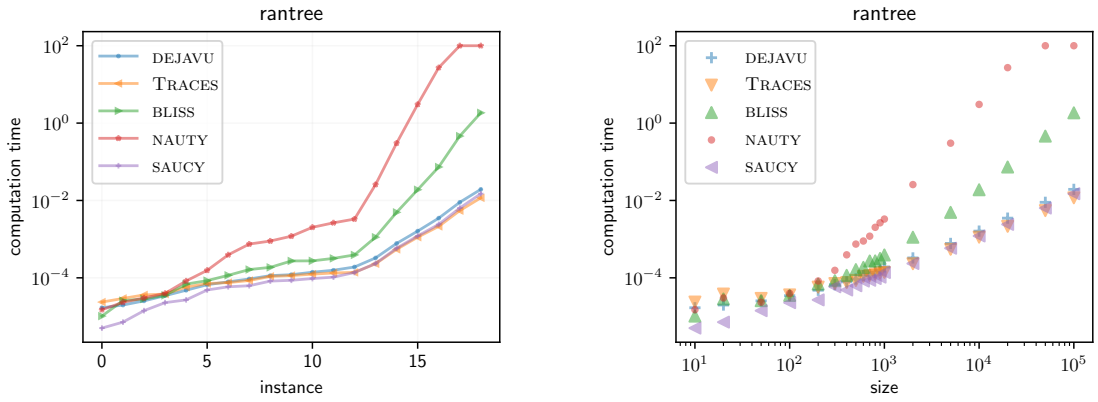
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.33: Benchmark results for the graph class `ran10`.



(a) Sorted according to runtime. (b) Sorted according to size.

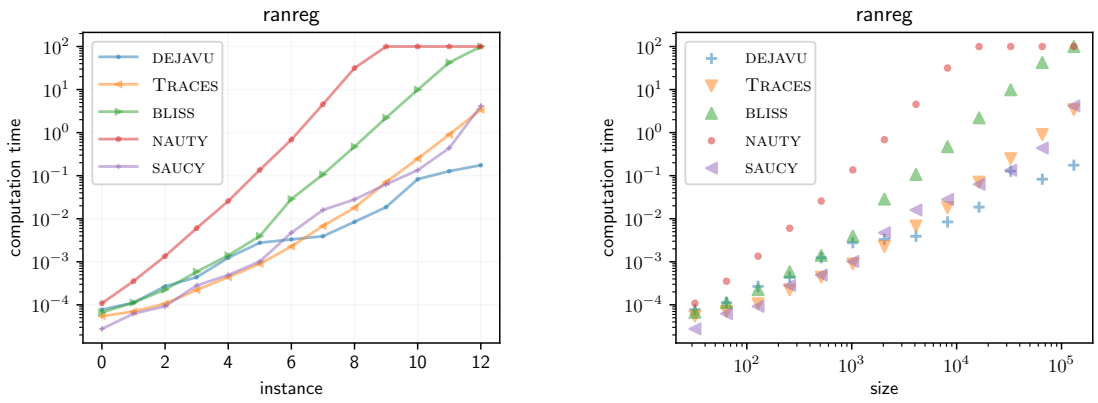
Figure 7.34: Benchmark results for the graph class `ransq`.



(a) Sorted according to runtime.

(b) Sorted according to size.

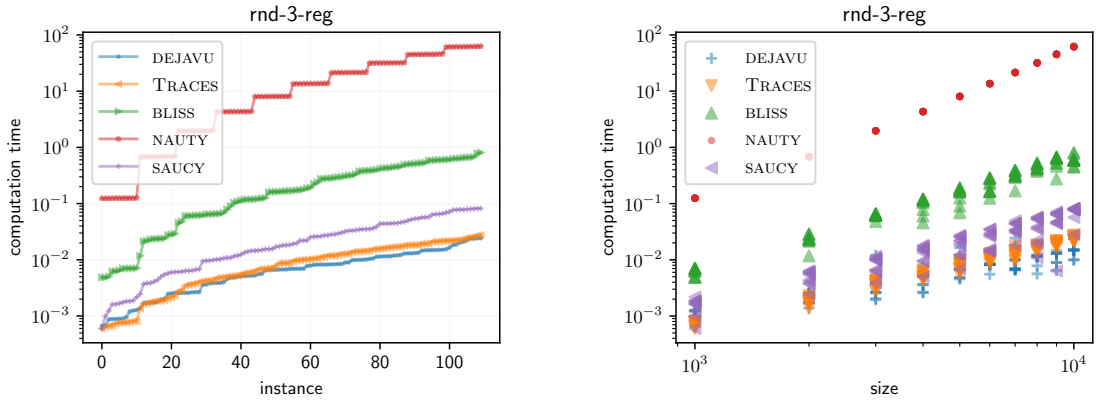
Figure 7.35: Benchmark results for the graph class `rantree`.



(a) Sorted according to runtime.

(b) Sorted according to size.

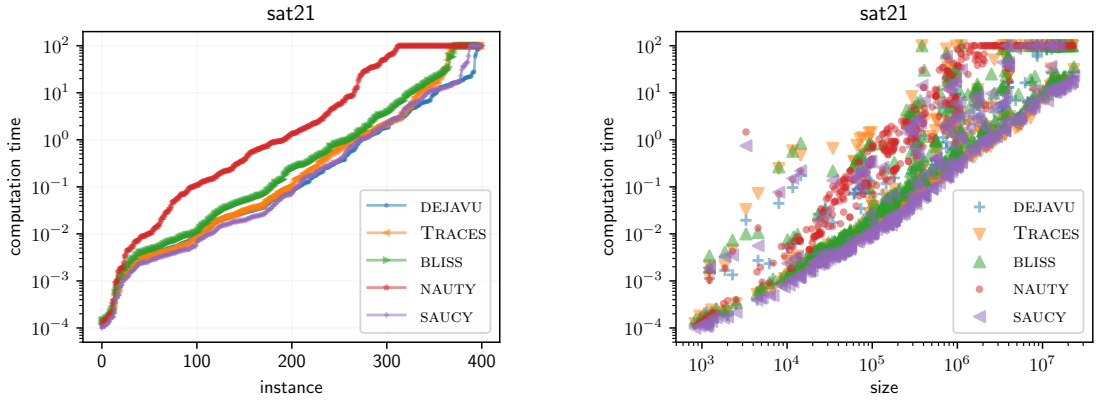
Figure 7.36: Benchmark results for the graph class `ranreg`.



(a) Sorted according to runtime.

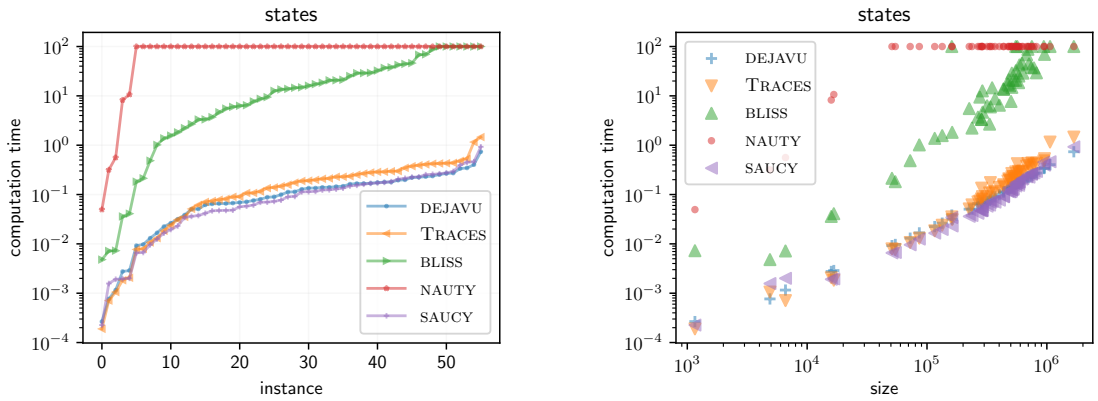
(b) Sorted according to size.

Figure 7.37: Benchmark results for the graph class `rnd-3-reg`.



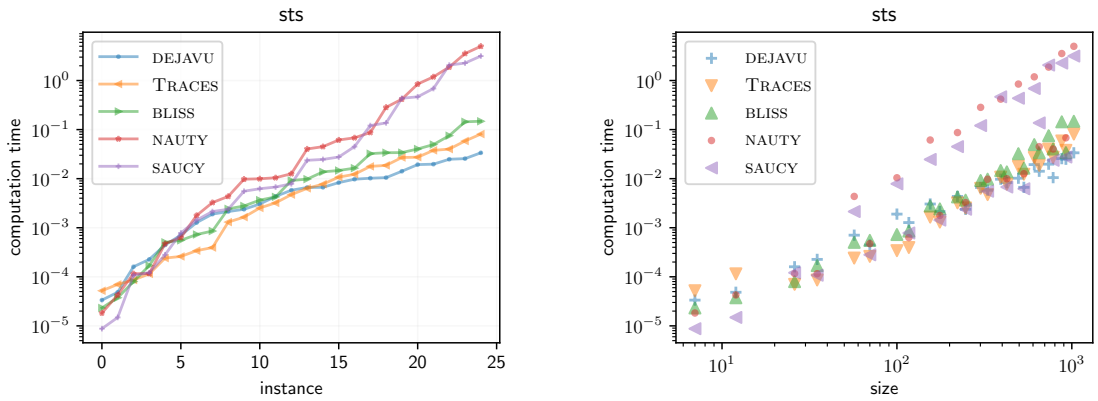
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.38: Benchmark results for the graph class `sat21`.



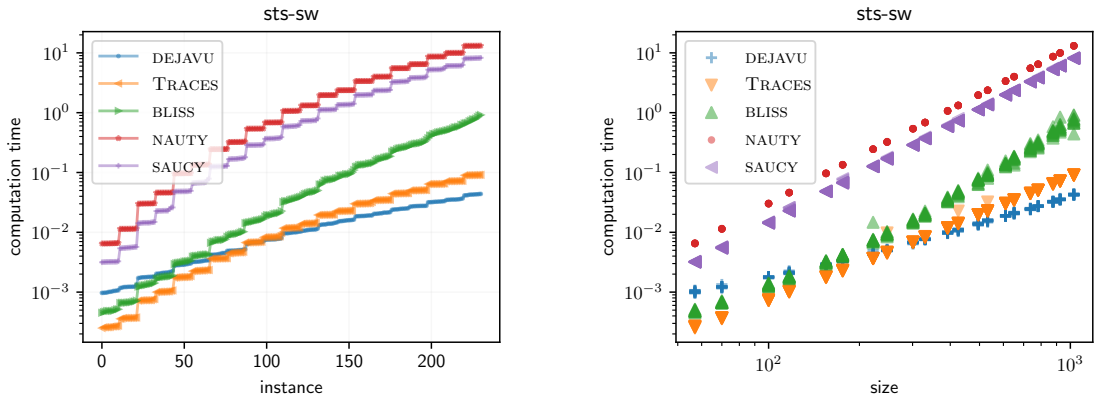
(a) Sorted according to runtime. (b) Sorted according to size.

Figure 7.39: Benchmark results for the graph class `states`.



(a) Sorted according to runtime. (b) Sorted according to size.

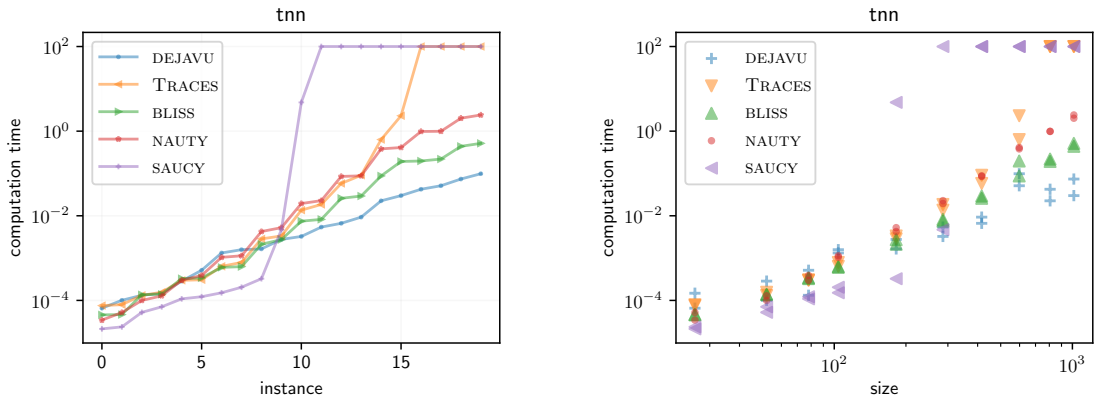
Figure 7.40: Benchmark results for the graph class `sts`.



(a) Sorted according to runtime.

(b) Sorted according to size.

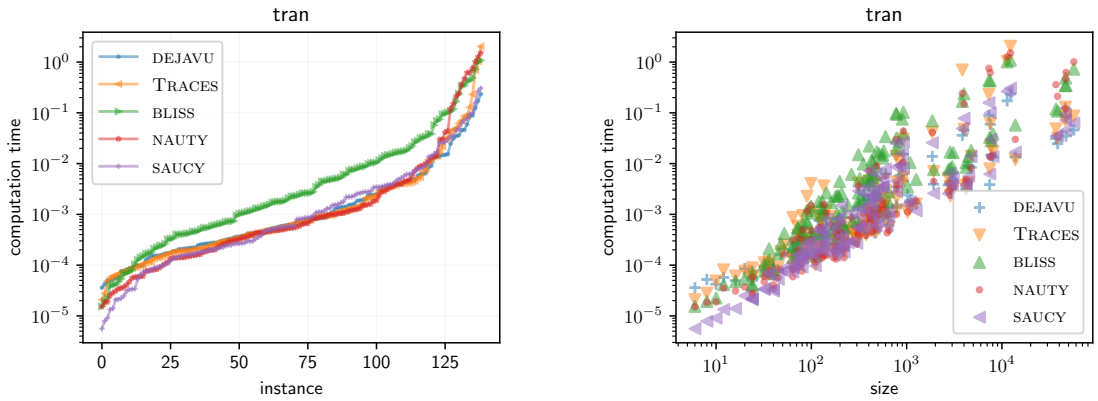
Figure 7.41: Benchmark results for the graph class `sts-sw`.



(a) Sorted according to runtime.

(b) Sorted according to size.

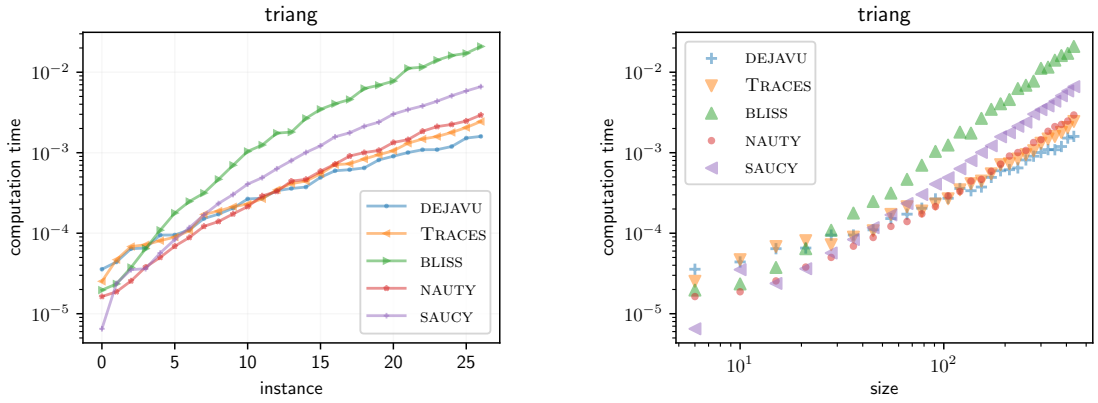
Figure 7.42: Benchmark results for the graph class `tnn`.



(a) Sorted according to runtime.

(b) Sorted according to size.

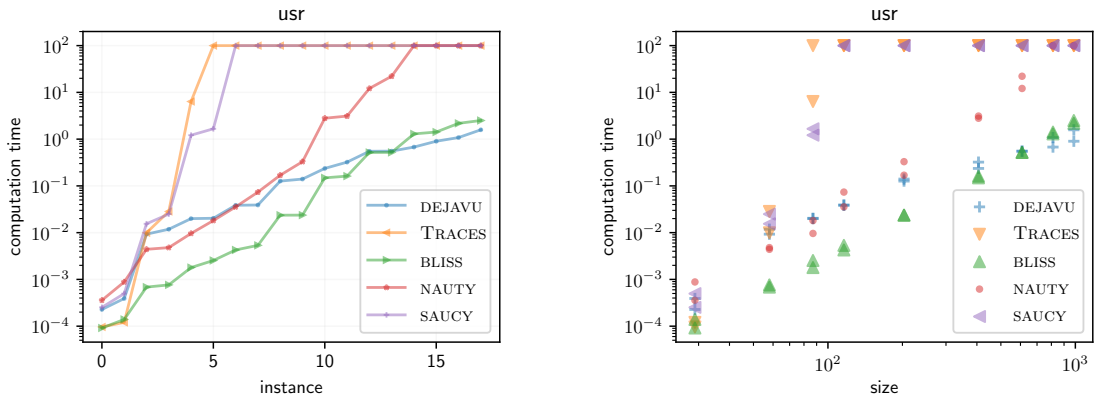
Figure 7.43: Benchmark results for the graph class `tran`.



(a) Sorted according to runtime.

(b) Sorted according to size.

Figure 7.44: Benchmark results for the graph class `triang`.



(a) Sorted according to runtime.

(b) Sorted according to size.

Figure 7.45: Benchmark results for the graph class `usr`.

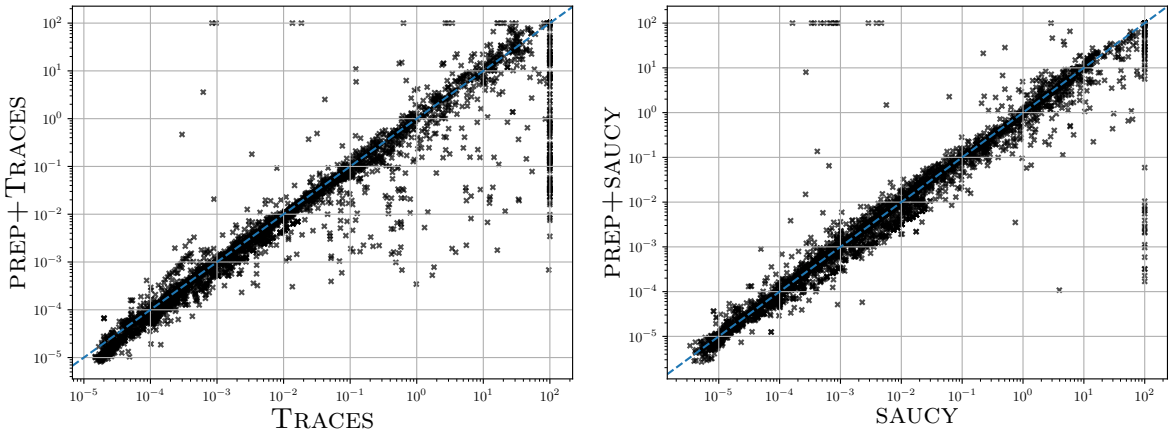
7.3 Preprocessing

We test the solvers NAUTY, SAUCY, BLISS, and TRACES in conjunction with our preprocessor described in Chapter 5. Our goal is to gauge whether the preprocessor speeds up the solvers. Considering the techniques of the preprocessor, the hope is that solvers become faster on large practical graphs, while not slowing down too much on difficult combinatorial graphs. We simply ran the solvers again on the entire benchmark suite, while applying the preprocessor first.

Figure 7.46 shows the results of our benchmarks. Clearly, for NAUTY, BLISS, and TRACES we can observe a significant overall speedup. However, on average, the preprocessor is also beneficial for SAUCY, albeit this is not as clear as for the other solvers. Considering the design of SAUCY however, this is to be expected. The preprocessor speeds up NAUTY on 80.1% of instances, SAUCY on 69.2% of instances, BLISS on 73.4% of instances, and TRACES on 72.8% of instances. The number of timeouts is reduced for all solvers.

It should be noted that some combinatorial instances are solved considerably slower using the preprocessor. (Others get faster.) In particular, in many cases, this is despite the fact that the preprocessor does not manipulate the graph structurally. The change in performance can however be explained as follows: the preprocessor applies color refinement, and passes the already refined graph to the solver. While every solver of course would also apply color refinement internally, the ordering of the colors may be different when done by the preprocessor. In turn, this can have knock-on effects for other strategies: the cell selector may now indeed choose different cells, which can lead to large differences in the IR tree itself.

Overall, it does however seem that the application of the preprocessor is generally beneficial for all solvers.



(a) TRACES versus preprocessor+TRACES.

(b) SAUCY versus preprocessor+SAUCY.

(c) BLISS versus preprocessor+BLISS.

(d) SAUCY versus preprocessor+NAUTY.

Figure 7.46: Testing state-of-the-art solvers using the preprocessor on all tested graphs. The y-axis is the runtime (in seconds) of the solver additionally using the DEJAVU preprocessor, and the x-axis is the runtime of the solver without the use of the preprocessor. Points below the diagonal indicate the use of the preprocessor is beneficial.

Chapter 8

Conclusions and Outlook

We conclude this thesis with a short summary of our results, potential future work, as well as open problems.

8.1 Conclusions

This exposition began with an introduction to the IR paradigm, as well as current state-of-the-art solvers implemented within the IR paradigm.

We found that a distinct feature of the state-of-the-art solver Traces is its unique traversal strategies for the IR backtracking trees, which seems to enable its outstanding performance on difficult combinatorial graphs. This prompted our theoretical investigation into IR backtracking strategies. First, our theoretical model offered an explanation for the effectiveness of the Traces strategy. Second, we found near-optimal Monte Carlo and Las Vegas strategies within the model, which asymptotically outperform all traversal strategies currently used in practice in the worst-case.

We then investigated *color refinement*, a crucial building block of IR algorithms. First, we gave a description of a state-of-the-art color refinement implementation, further arguing its correctness and worst-case runtime. Second, we modeled a central design choice of the algorithm: the worklist order, i.e., the order in which color classes are refined. Within a theoretical online and offline model, we showed that no worklist order is competitive beyond a logarithmic factor to every other worklist order.

Turning to the practical side of things, we then described our *universal preprocessor* for symmetry detection. The preprocessor is designed to work with all state-of-the-art symmetry detection tools. We described techniques to reduce vertices of low degree, twins, as well as further reduction techniques based on the quotient graph. Benchmarks demonstrated that the preprocessor speeds up all existing state-of-the-art solvers.

Finally, we described the dejavu algorithm for symmetry detection. The algorithm is heavily based on our theoretical investigation: in particular, its main backtracking strategy is based on the near-optimal Monte Carlo traversal strategy. We further augmented the Monte Carlo strategy with many novel practical techniques. The motivation behind all of these augmentations was to enable the solver to work well across a wide range of graph classes. Benchmarks then demonstrated that dejavu outperforms all state-of-the-art solvers on the majority of tested graph classes while being close to the fastest solver on almost all classes. In particular, dejavu seems to gain an advantage as graphs become more difficult or larger. On the other hand, dejavu slightly lacks performance on very easy or small graphs, at least compared to some of the other solvers. Overall,

our benchmarks provide evidence that DEJAVU is indeed the most powerful symmetry detection algorithm currently in use.

8.2 Future Work

There seem to be many promising directions in which the present work, as well as the practical implementation, could be expanded upon.

Canonical Labeling. The present algorithm solves symmetry detection and does *not* solve canonical labeling. An interesting approach could be to run DEJAVU first and then a tailored canonical labeling algorithm that can efficiently use a given automorphism group. A similar approach has already been explored using SAUCY and BLISS in [61], where SAUCY and BLISS are essentially applied in lock-step. However, we believe the approach could be even more successful using DEJAVU and a canonical labeling approach akin to TRACES. Furthermore, the preprocessing routines could be expanded to canonical labeling.

Las Vegas in Practice. Another interesting approach could be to explore the Las Vegas strategy of Chapter 3 in practice. Again, we could simply run DEJAVU as-is first and then only apply the Las Vegas algorithm whenever DEJAVU does not terminate deterministically. Furthermore, random walks could most likely already be used to determine a fairly balanced split.

Luks' Framework in Practice. In general, it seems difficult to apply techniques from computational group theory as used in theoretical algorithms for graph isomorphism in practice. However, in DEJAVU, applying techniques only under specific circumstances, e.g., if previous techniques already amortize them, is straightforward. In particular, the algorithm for bounded color multiplicity could be of interest in practice [71], since it could solve the current class of worst-case examples [86, 87] in polynomial time.

Canonical Labeling Traversal. Our model for IR tree traversal does not seem to properly model canonical labeling. The author does not believe that canonical labeling can be solved in time that is sublinear in the size of the IR backtracking tree. However, in order to prove this, a refined theoretical model seems to be necessary.

Incremental Graph Isomorphism. In certain applications, it would be interesting to be able to compute the symmetries of a graph, modify it slightly, and then compute the symmetries again (e.g., to compute local symmetries in a backtracking tree). While adding or removing a single edge can completely discretize an otherwise symmetrical graph, in practice, there could still be many cases in which the incremental problem can be solved efficiently.

SIMD and Color Refinement. Our implementation of color refinement may still leave room for significant improvements. An interesting case could be small or very dense graphs. Here, using single instruction multiple data (SIMD) instructions could potentially greatly improve performance.

Certified Non-Isomorphism. While we can efficiently certify isomorphisms and automorphisms of graphs, this does not necessarily hold true for non-isomorphism, non-automorphism, and canonical labeling. As mentioned in Chapter 1, using a randomized protocol, non-isomorphism and non-automorphism can be verified. Still, it could be of interest to enhance solvers to provide a certificate for non-isomorphism or for canonical labelings, which can be checked using a machine-verified checker [73]. A particular case in which this could be interesting is canonical labeling: if canonical labeling is used as part of a larger process (e.g., exhaustive graph generation), this could, in turn, help provide a certificate for said larger process.

For all intents and purposes, it seems that practical graph isomorphism is still not "solved" and is sure to be pushed further in the future.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Shatter: efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 836–839. ACM, 2003.

[3] Markus Anders, Jendrik Brachter, and Pascal Schweitzer. A characterization of individualization-refinement trees. In *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 24:1–24:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[4] Markus Anders and Pascal Schweitzer. Engineering a fast probabilistic isomorphism test. In *Proceedings of the Symposium on Algorithm Engineering and Experiments, ALENEX 2021, Virtual Conference, January 10-11, 2021*, pages 73–84. SIAM, 2021.

[5] 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.

[6] Markus Anders and Pascal Schweitzer. Search problems in trees with symmetries: Near optimal traversal strategies for individualization-refinement algorithms. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 16:1–16:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

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

[8] Markus Anders, Pascal Schweitzer, and Florian Wetzels. Comparative design-choice analysis of color refinement algorithms beyond the worst case. In *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.

[9] Vladimir L. Arlazarov, I.I. Zuev, A.V. Uskov, and I.A. Faradzhev. An algorithm for the reduction of finite non-oriented graphs to canonical form. *USSR Computational Mathematics and Mathematical Physics*, 14(3):195–201, 1974.

[10] Vikraman Arvind, Frank Fuhlbrück, Johannes Köbler, Sebastian Kuhnert, and Gaurav Rattan. The parameterized complexity of fixing number and vertex individualization in graphs. In *MFCS2016*, volume 58 of *LIPIcs*, pages 13:1–13:14, 2016.

[11] Vikraman Arvind, Johannes Köbler, Gaurav Rattan, and Oleg Verbitsky. On tinhofer's linear programming approach to isomorphism testing. In *Mathematical Foundations of Computer Science 2015 - 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part II*, volume 9235 of *Lecture Notes in Computer Science*, pages 26–37. Springer, 2015.

[12] László Babai. Graph isomorphism in quasipolynomial time [extended abstract]. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 684–697. ACM, 2016.

[13] László Babai. Canonical form for graphs in quasipolynomial time: preliminary report. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 1237–1246. ACM, 2019.

[14] László Babai, Paul Erdös, and Stanley M. Selkow. Random graph isomorphism. *SIAM J. Comput.*, 9(3):628–635, 1980.

[15] László Babai and Shlomo Moran. Arthur-merlin games: A randomized proof system, and a hierarchy of complexity classes. *J. Comput. Syst. Sci.*, 36(2):254–276, 1988.

[16] 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.

[17] Armin Biere, Florian Lonsing, and Martina Seidl. Blocked clause elimination for QBF. In *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.

[18] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *J. Algorithms*, 11(4):631–643, 1990.

[19] Jendrik Brachter. *Combinatorial approaches to the group isomorphism problem*. PhD thesis, Technische Universität Darmstadt, Darmstadt, 2023.

[20] Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Comb.*, 12(4):389–410, 1992.

[21] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*, volume 5917 of *Lecture Notes in Computer Science*, pages 75–85. Springer, 2009.

[22] Duc-Hiep Chu and Joxan Jaffar. A complete method for symmetry reduction in safety verification. In *Proceedings of the 24th international conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 616–633. Springer, 2012.

[23] Paolo Codenotti, Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Conflict analysis and branching heuristics in the search for graph automorphisms. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 907–914. IEEE Computer Society, 2013.

[24] Michael Codish, Graeme Gange, Avraham Itzhakov, and Peter J. Stuckey. Breaking symmetries in graphs: The nauty way. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2016.

[25] Derek G. Corneil and Calvin C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.

[26] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *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.

[27] 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.

[28] 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.

[29] Anuj Dawar and Kashif Khan. Constructing hard examples for graph isomorphism. *J. Graph Algorithms Appl.*, 23(2):293–316, 2019.

[30] David Déharbe, Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Exploiting symmetry in SMT problems. In *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.

[31] dejavu. `https://automorphisms.org`, source code tested in this thesis is also archived at swh:1:dir:5d33400fd9149c6e3507cc1b1f9a0f3c396d6372.

[32] Jo Devriendt and Bart Bogaerts. BreakID: Static symmetry breaking for ASP (system description). *CoRR*, abs/1608.08447, 2016.

[33] Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In *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.

[34] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016.

[35] Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 49–56. IEEE Computer Society, 2012.

[36] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *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.

[37] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002.

[38] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kızıltan, Ian Miguel, and Toby Walsh. Matrix modelling. Technical Report APES-36-2001, APES group (2001), 2001.

[39] Billy J. Franks, Christopher Morris, Ameya Velingker, and Floris Geerts. Weisfeiler-leman at the margin: When more expressivity matters. *CoRR*, abs/2402.07568, 2024.

[40] Billy Joe Franks, Markus Anders, Marius Kloft, and Pascal Schweitzer. A systematic approach to universal random features in graph neural networks. *Transactions on Machine Learning Research*, 2023.

[41] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.13.0*, 2024.

[42] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006.

[43] 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.

[44] Patrice Godefroid. Exploiting symmetry when model-checking software. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*, volume 156 of *IFIP Conference Proceedings*, pages 257–275. Kluwer, 1999.

[45] Mark K. Goldberg. A nonfactorial algorithm for testing isomorphism of two graphs. *Discret. Appl. Math.*, 6(3):229–236, 1983.

[46] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[47] Martin Grohe. Equivalence in finite-variable logics is complete for polynomial time. In *FOCS '96*, pages 264–273. IEEE Computer Society, 1996.

[48] Martin Grohe. Fixed-point definability and polynomial time on graphs with excluded minors. *J. ACM*, 59(5):27:1–27:64, 2012.

[49] Martin Grohe and Daniel Neuen. Isomorphism for tournaments of small twin width. *CoRR*, abs/2312.02048, 2023.

[50] Martin Grohe and Pascal Schweitzer. Isomorphism testing for graphs of bounded rank width. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1010–1029. IEEE Computer Society, 2015.

[51] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[52] Marijn J. H. Heule. Optimal symmetry breaking for graph problems. *Math. Comput. Sci.*, 13(4):533–548, 2019.

[53] Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.

[54] Christopher Hojny and Marc E. Pfetsch. Polytopes associated with symmetry handling. *Math. Program.*, 175(1-2):197–240, 2019.

[55] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[56] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs (preliminary report). In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1974, Seattle, Washington, USA*, pages 172–184. ACM, 1974.

[57] Tommi A. Junttila, Matti Karppa, Petteri Kaski, and Jukka Kohonen. An adaptive prefix-assignment technique for symmetry reduction. *J. Symb. Comput.*, 99:21–49, 2020.

[58] 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.

[59] 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.

[60] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Conflict anticipation in the search for graph automorphisms. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 243–257. Springer, 2012.

[61] Hadi Katebi, Karem A. Sakallah, and Igor L. Markov. Graph symmetry detection and canonical labeling: Differences and synergies. In *Turing-100 - The Alan Turing Centenary, Manchester, UK, June 22-25, 2012*, volume 10 of *EPiC Series in Computing*, pages 181–195. EasyChair, 2012.

[62] 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.

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

[64] Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In *27th International Conference on Principles and Practice of Constraint Programming, CP*, volume 210 of *LIPIcs*, pages 34:1–34:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[65] Tuukka Korhonen, Jeremias Berg, Paul Saikko, and Matti Järvisalo. Maxpre: An extended maxsat preprocessor. In *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.

[66] Martin Kutz and Pascal Schweitzer. Screwbox: a randomized certifying graph-non-isomorphism algorithm. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. SIAM, 2007.

[67] Wenchao Li, Hossein Saidi, Huascar Sanchez, Martin Schäf, and Pascal Schweitzer. Detecting similar programs via the Weisfeiler-Leman graph kernel. In *Proceedings of the 15th International Conference on Software Reuse: Bridging with Social-Awareness*, volume 9679 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2016.

[68] Yanxi Liu, Hagit Hel-Or, Craig S. Kaplan, and Luc Van Gool. Computational symmetry in computer vision and computer graphics. *Foundations and Trends® in Computer Graphics and Vision*, 5(1–2):1–195, 2010.

[69] José Luis López-Presa and Antonio Fernández Anta. Fast algorithm for graph isomorphism testing. In *Experimental Algorithms, 8th International Symposium, SEA 2009, Dortmund, Germany, June 4-6, 2009. Proceedings*, volume 5526 of *Lecture Notes in Computer Science*, pages 221–232. Springer, 2009.

[70] José Luis López-Presa, Antonio Fernández Anta, and Luis Núñez Chiroque. Conauto-2.0: Fast isomorphism testing and automorphism group computation. *CoRR*, abs/1108.1060, 2011.

[71] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.

[72] François Margot. Symmetry in integer linear programming. In *50 Years of Integer Programming 1958-2008 - From the Early Years to the State-of-the-Art*, pages 647–686. Springer, 2010.

[73] Ross M. McConnell, Kurt Mehlhorn, Stefan Näher, and Pascal Schweitzer. Certifying algorithms. *Comput. Sci. Rev.*, 5(2):119–161, 2011.

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

[75] Brendan D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

[76] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

[77] Brendan D. McKay, Mehmet Aziz Yirik, and Christoph Steinbeck. Surge: a fast open-source chemical graph generator. *J. Cheminformatics*, 14(1):24, 2022.

[78] Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. Cdclsym: Introducing effective symmetry breaking in SAT solving. In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018.

[79] Alice Miller, Alastair F. Donaldson, and Muffy Calder. Symmetry in temporal logic model checking. *ACM Computing Surveys*, 38(3), 2006.

[80] Gary L. Miller. Graph isomorphism, general remarks. *J. Comput. Syst. Sci.*, 18(2):128–142, 1979.

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

[82] Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and Computation, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, June 7-10, 1995*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 239–256. DIMACS/AMS, 1995.

[83] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*, pages 4602–4609. AAAI Press, 2019.

[84] A.G. Munford. A note on the uniformity assumption in the birthday problem. *Amer. Statist.*, 31(3):119, 1977.

[85] nauty and Traces. `http://pallini.di.uniroma1.it`.

[86] Daniel Neuen and Pascal Schweitzer. Benchmark graphs for practical graph isomorphism. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 60:1–60:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[87] Daniel Neuen and Pascal Schweitzer. An exponential lower bound for individualization-refinement algorithms for graph isomorphism. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2018, Los Angeles, CA, USA, June 25-29, 2018*, pages 138–150. ACM, 2018.

[88] William P. Orrick. Switching operations for hadamard matrices. *SIAM J. Discret. Math.*, 22(1):31–50, 2008.

[89] James Ostrowski, Jeff T. Linderoth, Fabrizio Rossi, and Stefano Smriglio. Constraint orbital branching. In *Integer Programming and Combinatorial Optimization, 13th International Conference, IPCO 2008, Bertinoro, Italy, May 26-28, 2008, Proceedings*, volume 5035 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2008.

[90] PACE challenge. `https://pacechallenge.org/`.

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

[92] 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.

[93] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). *CoRR*, abs/0804.4881, 2008.

[94] Adolfo Piperno. Isomorphism test for digraphs with weighted edges. In *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.

[95] Milan Randić. On canonical numbering of atoms in a molecule and graph isomorphism. *Journal of Chemical Information and Computer Sciences*, 17(3):171–180, 1977.

[96] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 475–484. ACM, 1997.

[97] Ronald C. Read and Derek G. Corneil. The graph isomorphism disease. *J. Graph Theory*, 1(4):339–363, 1977.

[98] David J. Rosenbaum. Breaking the $n^{\log n}$ barrier for solvable-group isomorphism. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013, New Orleans, Louisiana, USA, January 6-8, 2013*, pages 1054–1073. SIAM, 2013.

[99] Irene Luque Ruiz and Miguel Ángel Gómez-Nieto. A java tool for the management of chemical databases and similarity analysis based on molecular graphs isomorphism. In *Computational Science – ICCS 2008*, pages 369–378, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[100] Ashish Sabharwal. Symchaff: exploiting symmetry in a structure-aware satisfiability solver. *Constraints An Int. J.*, 14(4):478–505, 2009.

[101] Karem A. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 509–570. IOS Press, 2021.

[102] SAT Competition 2021. `https://satcompetition.github.io/2021/`.

[103] Pascal Schweitzer. *Problems of unknown complexity: graph isomorphism and Ramsey theoretic numbers*. Phd. thesis, Universität des Saarlandes, Saarbrücken, Germany, 2009.

[104] Ákos Seress. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press, 2003.

[105] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[106] Charles C. Sims. Computational methods in the study of permutation groups. In *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon, 1970.

[107] Stoicho D. Stoichev. New exact and heuristic algorithms for graph automorphism group and graph isomorphism. *ACM J. Exp. Algorithmics*, 24(1):1.15:1–1.15:27, 2019.

[108] Greg Tener. *Attacks On Difficult Instances Of Graph Isomorphism: Sequential And Parallel Algorithms*. PhD thesis, University of Central Florida, 2010.

[109] Alasdair Urquhart. Hard examples for resolution. *J. ACM*, 34(1):209–219, 1987.

List of Figures

List of Algorithms

List of Tables

Academic Curriculum Vitae

2013-2014	*DHBW Mannheim*
2014-2017	*TU Kaiserslautern* Bachelor of Science, Computer Science
2017-2019	*TU Kaiserslautern* Master of Science, Computer Science
2019-2021	*TU Kaiserslautern* PhD Student, Supervisor: Pascal Schweitzer
2021-2024	*TU Darmstadt* PhD Student, Supervisor: Pascal Schweitzer