Problems of Unknown Complexity

Graph isomorphism and Ramsey theoretic numbers

Dissertation zur Erlangung des Grades
des Doktors der Naturwissenschaften (Dr. rer. nat.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

vorgelegt von

Pascal Schweitzer

Saarbrücken
2009

Revised April 2012.

Tag des Kolloquiums: 24. Juli 2009:

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Professor Dr. Joachim Weickert

Berichterstatter:
Professor Dr. Kurt Mehlhorn, Max-Planck-Institut für Informatik, Saarbrücken
Professor Dr. Markus Bläser, Fachbereich Informatik der Universität des Saarlandes
Professor Dr. Brendan McKay, Department of Computer Science, Australian National University

Mitglieder des Prüfungsausschusses:

Professor Dr. Joachim Weickert	(Vorsitzender)
Professor Dr. Kurt Mehlhorn	
Professor Dr. Markus Bläser	
Professor Dr. Brendan McKay	(via Videokonferenz)
Dr. Nicole Megow	

In memory of Martin Kutz

Zusammenfassung

Wir entwickeln Algorithmen für drei kombinatorische Probleme unbekannter Komplexität: Das Graphisomorphieproblem, die Berechnung von van der Waerden-Zahlen und die Berechnung von Ramsey-Zahlen. Mit theoretischen und praktischen Methoden wird ein Vergleich zu bereits existierenden Algorithmen gezogen.

Der Schraubenkasten, ein zertifizierender, randomisierter Graph-nicht-Isomorphie Algorithmus, führt zufällige Stichproben in zwei gegeben Graphen durch, und schließt durch Festellen von statistisch signifikant abweichendem Verhalten der gesammelten Daten auf Nichtisomorphie. Die Durchführung der Stichproben und damit die erhaltenen Daten sowie die verwendete Methode zum Festellen statistisch abweichenden Verhaltens passen sich dabei den Eingabegraphen an. Auf isomorphen Graphen wird mit hoher Wahrscheinlichkeit ein Isomorphismus gefunden, der als Zertifikat dient. Für nichtisomorphe Eingabegraphen dienen als randomisiertes Zertifikat die Zusammensetzung des Schraubenkastens und die Spezifizierung eines günstigen statistischen Tests.

Zur Berechnung von van der Waerden-Zahlen entwickeln wir einen Algorithmus, der durch die Verwendung von Platzhaltern ein gleichzeitiges Bearbeiten von verschiedenen Elementen des zu durchsuchenden Lösungsraums ermöglicht. Mit ihm werden neue van der Waerden-Zahlen berechnet.

Der Zusammenhang der ersten beiden Probleme ist durch das dritte gegeben, dessen Lösung die anderen Lösungen zu einem Algorithmus verknüpft, der Ramsey-Zahlen berechnet.

Abstract

We consider three computational problems with unknown complexity status: The graph isomorphism problem, the problem of computing van der Waerden numbers and the problem of computing Ramsey numbers. For each of the problems, we devise an algorithm that we analyze with theoretical and practical means by a comparison with contemporary algorithms that solve the respective problems.

The ScrewBox algorithm solves the graph isomorphism problem by a random sampling process. Given two graphs, the algorithm randomly searches an invariant that may be randomly evaluated quickly and that shows significant statistical difference on the input graphs. This invariant is gradually and adaptively constructed depending on the difficulty of the input. Isomorphism is certified by supplying an isomorphism. Non-isomorphism is certified by the ScrewBox, the invariant whose statistical behavior deviates on the input graphs, together with the appropriate statistical test.

The wildcards algorithm for van der Waerden numbers solves the second problem. Its key technique is to treat colorings of integers avoiding monochromatic arithmetic progressions simultaneously by allowing ambiguity. This, together with a specific preprocessing step, forms the algorithm that is used to compute previously unknown van der Waerden numbers.

The wildcards algorithm for Ramsey numbers combines the techniques and algorithms with which we approach the first two problems to solve the third problem.

Contents

1 Introduction

When we are posed the algorithmic question: "How do you compute this efficiently?" with classical computational complexity theory we produce two kinds of answers: Either we devise a provably efficient algorithm or we show, using the theoretical framework, that it is likely, that we will never be able to construct an efficient algorithm for the problem, no matter how hard we try. Despite the large applicability of the classical computational complexity, there are problems for which our tools have not provided either answer yet. The complexity of these problems is unknown.

This thesis is concerned with three fundamental computational problems, arising from combinatorics, with unknown complexity status:

1. the graph isomorphism problem,
2. the computation of van der Waerden numbers and
3. the computation of Ramsey numbers.

For decades, numerous approaches have been taken to each of the problems in order to show one of the classical alternatives. Still, the unsettled complexity status of the problems remains. Nevertheless, we desire algorithms that solve these problems as efficiently as possible.

In this thesis we develop algorithmic concepts that address the problems. Using the concepts, for each problem we design an algorithm that we evaluate by means of theoretical and practical comparison to state-of-the-art algorithms that have previously been designed. The design of such algorithms goes hand in hand with mathematical insight into the combinatorial structures involved in the problems and into their complexity.

The third problem is strongly connected to the first two problems and thus forms the link between the two. This becomes apparent, as the algorithms devised for the graph isomorphism problem and the problem of computing van der Waerden numbers are merged to form an algorithm that computes Ramsey numbers. We briefly describe the three problems considered.

Graph isomorphism

Numerous graph theoretical treatises mention Euler's famous problem, the Seven Bridges of Königsberg [39], in their introduction. In 1736 Euler asks whether it is possible to tour Königsberg, using all of its bridges exactly once. The solution abstracts the paths within the city into a graph, consisting of vertices, the islands, and edges between them, the bridges. The graph thus models the relation between vertices that governs whether there is an edge between them. Readily agreed by Euler and his readers, for the existence of such a tour, the names of the different islands and their geographic location is irrelevant. Mathematically this abstraction is reformulated to the fact that our solution depends only on the isomorphism type of the graph. Informally, two graphs are isomorphic if we can not differentiate them, after we ignore labels (names) for vertices and edges and only consider the relation governed by the edges. Thus we abstract the information, that we can reach the Kneiphof Island from the Altstadt via the Krämerbrücke or from the Vorstadt via the Green Bridge, but cannot reach the Altstadt directly from the Vorstadt, to a graph on three vertices that has two edges. In a different scenario where we model whether people know each other, we also abstract the information, that Agate knows Boris and Ceceilia, but Boris and Ceceilia do not know each other, to a graph on three vertices that has two edges. We say, that the graphs representing the two examples are isomorphic. The algorithmic task of deciding whether two graphs are isomorphic becomes increasingly difficult as the number of vertices and the number of edges increase. When considering the road network as an example of a graph that contains a lot of information, we see that computers are indispensable when modeling large graphs. Besides its applications in the natural sciences, where it is for example used to identify chemical compounds [111], the graph isomorphism problem has applications in mathematics and computer science. It is crucial for enumeration of various combinatorial objects, such as Ramsey graphs [93, 94], and used to compare circuit layouts against their specification [35]. Currently, graph isomorphism is a candidate for a quantum algorithm [103]. In the traditional theory of computational complexity, the problem is a reappearing candidate with various unusual properties [72].

Van der Waerden numbers

Van der Waerden's theorem [126], which also proves the existence of the van der Waerden numbers, is a Ramsey theoretic result. Roughly speaking, Ramsey theoretic results state that in large structures which are partitioned into finitely many parts, a certain smaller substructure must emerge within one of the parts. Van der Waerden's theorem in particular states, that when the integers are partitioned into finitely many sets, then one of these sets must contain arbitrarily long sequences of equidistant integers. Such a sequence, for example $3, 7, 11, 15, \ldots$, is called an arithmetic progression.

The van der Waerden numbers are quantifications of the theorem, in the sense that they determine the size of the smallest subset of the integers $\{1, \ldots, n\}$ for which these arithmetic progressions arise, whenever the set is partitioned arbitrarily into a fixed number of parts. We are concerned with the computation of the van der Waerden numbers.

As demonstrated in Rosta's dynamic survey [112], the applications of Ramsey theory are numerous within numerous fields of mathematics. The survey includes various applications of van der Waerden's theorem, including applications in number theory, lower bound constructions for the computation of boolean functions and in finite model theory. Recently, connections between the computation of van der Waerden numbers and propositional theories have been shown [33]. Using this connection van der Waerden numbers have been computed and at the same time they serve as benchmarks for solvers of satisfiability problems.

Ramsey numbers

In contrast to van der Waerden's theorem, which deals with partitions of integers, Ramsey's theorem [110] deals with partitions of edges contained in a complete graph. It shows for example, that among 6 people either 3 are pairwise strangers or 3 of the people all know each other. In its generality, the theorem states that however we partition the edges of a sufficiently large complete graph into finitely many sets, we find a large complete subgraph, whose edges are all contained in the same partition class. The Ramsey numbers describe how large the original graph must be, so that we can find these subgraphs of a certain size. It is not known how fast these numbers grow asymptotically. In his essay on the two cultures of mathematics, Gowers [48] mentions this problem as "one of the major problems in combinatorics" since "a solution to this problem is almost bound to introduce a major new technique." As direct application Boppana and Halldórsson [16, 17] use Ramsey numbers to devise a polynomial time approximation algorithm that finds a large complete subgraph of a guaranteed, but not necessarily optimal, size.

Contribution

In this thesis we develop algorithms for each of the three combinatorial problems and evaluate them by theoretical and practical comparison with the state-of-the-art.

Our first algorithm, the ScrewBox, solves the graph isomorphism problem by randomly sampling within a pair of input graphs, and decides isomorphism by performing a statistical test. The algorithm aims in particular at a fast detection of non-isomorphic inputs that appear similar. Given two graphs, the ScrewBox randomly searches an invariant that may be randomly evaluated quickly and that shows significant statistical difference on the two input graphs. This invariant is gradually and adaptively constructed, depending on the difficulty of the input.

We show with theoretical and practical means that the ScrewBox can compete with Nauty [88, 92], the benchmark algorithm for graph isomorphism. Nauty is based on the individualization refinement technique, generally considered as the fastest technique for isomorphism solvers available [9]. We show that the expected number of samplings performed by the sampling approach, which directly corresponds to the running time of a specific version of the ScrewBox algorithm, is at most the number of search tree nodes visited by the Nauty's individualization refinement approach. We further show on the theoretical side, that the ScrewBox easily handles the Cai-Fürer-Immerman construction, the most prominent method to produce pairs of non-isomorphic graphs that are difficult to distinguish for various current graph isomorphism algorithms. We develop a particular family of graph isomorphism invariants that is well suited for the ScrewBox's sampling approach. Practically we show that for a specific family of graphs that arise by combinatorial construction, non-isomorphism detection is infeasible for Nauty. In contrast ScrewBox is able to show non-isomorphism for these graphs.

The ScrewBox algorithm is a Monte Carlo algorithm with 1-sided error: If the input graphs are non-isomorphic graphs, the algorithm determines so. If the input graphs are isomorphic, the algorithm finds with a chosen probability of error an isomorphism. The ScrewBox uses a novel approach to graph isomorphism: It performs statistical tests to conclude an answer.

We demonstrate, again theoretically and practically, what kind of test is favorable for the algorithm, and show how such a test can be chosen efficiently. The distinctive feature of the statistical tests is the adaption to the changing behavior of the unknown distributions of the outcomes produced by the random samplings in the graphs. This is achieved by applying a filter that changes over time.

Exemplary, we outline three subproblems arising in the implementation of the ScrewBox algorithm and the algorithm engineering that has been performed to solve them efficiently. Besides answering whether the input graphs are isomorphic, the ScrewBox algorithm also provides the user with witness that certifies the output. In case the two input graphs are isomorphic, this witness is an isomorphism. In case the input graphs are non-isomorphic, the witness is checkable with a statistical test. To relate randomly checkable witnesses to the theory of certifying algorithms, we develop the concept of random certificates for Monte Carlo algorithms, i.e., for algorithms that err.

The wildcards algorithm, the second new algorithm presented in this thesis, computes mixed van der Waerden numbers. It is based on a technique that reduces the search space and is a generalization of a variant of delayed evaluation to more than two colors. The partitions of the integers considered in van der Waerden's theorem are usually considered as a colorings, with the parts corresponding to the color classes. The general idea behind the algorithm is to treat colorings of integers that avoid monochromatic arithmetic progressions simultaneously by allowing ambiguity in the colorings generated by the algorithm. This idea, together with a preprocessing step that further prunes the search space, composes an algorithm that was used, with one exception, to recompute all known mixed van der Waerden numbers. Moreover, two previously unknown van der Waerden numbers, $w(2,3,14;3) = 202$ and $w(2,2,3,11;4) = 141$, have been computed with the wildcards algorithm. The original implementation of the algorithm contained an error, and consequently, the value of $w(2,3,14;3)$ was erroneously computed to be 201. I thank Michal Kouril for pointing out this error to me and supplying me with an example of a colored sequence proving the original computation to be faulty. The original implementation has been revised and the computations have been repeated.

The third algorithm merges the previous two into a combined algorithm that computes Ramsey numbers. Its central aspects are the simultaneous treatment of colored graphs, in analogy to the simultaneous treatment of colorings of integers, and the isomorphism detection of the colored graphs that are obtained. The algorithm shows the connection between the techniques and algorithms with which the first two problems are approached. We highlight the search space reduction obtained with the wildcards algorithm for Ramsey numbers and show the difficulties that arise when this algorithm is intended for the computation of a new Ramsey number.

All three algorithms are intended as practical algorithms, solving prominent problems of unknown complexity. The theoretical comparisons and the computations performed on actual instances serve as proof of concept. The implementations of the algorithms have been thoroughly tested on various inputs. The emphasis of the implementation was set on the development of the concepts of and for the algorithms. Although the main focus of the implementation did not lie on algorithm engineering, a reasonable amount of it was necessary to process input instances of relevant size. Still, there is room for improvement of the efficiency of the implementation, to optimally exploit the techniques that were developed. The implementation of three algorithms is available at [116].

The thesis is written as a coherent, self-contained document. Its aim is to develop the theory required to understand the algorithms that were designed and to relate them to current research. The thesis is arranged in three chapters, each of which treats one of the three combinatorial problems.

Acknowledgements

I thank Kurt Mehlhorn for his continuing, motivating supervision and support. Further, I appreciate the inspiring and encouraging conversations I had with Petteri Kaski. I am thankful to Diane Tremor, for providing excellent command of the English language, and to Daniel Johannsen who always had an open ear that listened to my fragmentary thoughts and an observant eye that corrected my drafts.

Finally, I am greatly in debt and most thankful to my brother Patrick Schweitzer, who persistently improved the quality of my dissertation in virtually every aspect.

2 Graph isomorphism

The computational complexity of graph isomorphism has remained unresolved for over thirty years. No polynomial-time algorithm deciding whether two given graphs are isomorphic is known; neither could this problem be shown to be $\mathcal{NP}$-complete. Graph isomorphism is one of the two remaining open problems from Garey and Johnson's famous list [45] of computational problems with this unsettled complexity status.

The approaches taken to and the publications on graph isomorphism are numerous, (for an overview see [72]). While research is conducted on complexity issues for the problem in general, algorithms efficient on restricted graph classes are designed. In contrast to "typical" problems known to be $\mathcal{NP}$-complete, it is not easy to devise truly difficult graph isomorphism instances. The leading graph-isomorphism solver Nauty [88, 92] (see Section 2.2) easily finds isomorphisms for most graphs with several thousand vertices. Only highly structured graphs pose a real challenge for this program (see Section 2.8). These difficult instances usually arise from combinatorial constructions. Among the hardest known instances are point-line incidence graphs of finite projective planes.

Even though the problem remains open, larger and larger insight has been gained over the years. We focus on certain important concepts that have arisen over time. First (in Section 2.1) we set a framework of definitions and formally describe the graph isomorphism problem. After this we turn to three prominent algorithms available, namely McKay's Nauty (in Section 2.2), the Weisfeiler-Lehman algorithm (in Section 2.3) and Luks' bounded degree algorithm (in Section 2.5). In between (in Section 2.4) we present the graph construction devised by Cai, Fürer and Immerman, that constructs pairs of non-isomorphic graphs, which the Weisfeiler-Lehman algorithm fails to differentiate. We also present the application of the construction by Miyazaki, used to produce graphs on which Nauty has exponential running time.

We then introduce ScrewBox (in Section 2.6), a randomized non-isomorphism algorithm, and explain the statistical test employed by this algorithm (in Section 2.7). This algorithm takes a new algorithmic approach to graph isomorphism. It computes randomized certificates for *non*-isomorphism of pairs of graphs. Based on heuristic sampling rules, we search for substructures in pairs of given graphs to find statistical evidence for non-isomorphism. After we treat combinatorial graph constructions that yield challenging input pairs (in Section 2.8), we supply various details that are elsewhere omitted (in Section 2.9). We then evaluate the ScrewBox algorithm from a theoretical and a practical perspective (in Section 2.10). We show that on various graphs the algorithm is able to compete with the benchmark isomorphism solver Nauty, and show adequate performance on particular "difficult" instances, which are infeasible for the other solvers. We conclude with a view on deterministic and randomized certification (in Section 2.11).

We start with definitions and the description of the graph isomorphism problem.

2.1 The graph isomorphism problem

The central definition in the context of graph isomorphism, and of this chapter, is the concept of a graph:

Definition 1 (graph). A *simple undirected graph* G is a pair of sets (V, E) called vertices and edges respectively, such that the edges form a subset of the two-element subsets of the vertices: $E \subseteq \{\{v, v'\} \mid v, v' \in V\}$.

The vertices are also referred to as *nodes*. Two vertices $v, v' \in V$ that form an edge $\{v, v'\}$ are *neighbors* and are said to be *adjacent*. If we require the possibility for multiple edges between the same pair of vertices, we allow E to be a multiset of pairs of V. Such a graph is called a *multigraph*. An edge of the form $\{v, v\}$ with $v \in V$ is called a *loop*. If loops are absent, i.e., if the binary relation on V induced by E is anti-reflexive, we call the graph *loopless*. In the class of *directed* graphs the edge set consists of ordered pairs of vertices, i.e., $E \subseteq \{(v, v') \mid v, v' \in V\}$. Such a graph corresponds to an *undirected* graph if the binary relation is symmetric, i.e., if for all $(v, v') \in E$ we have $(v', v) \in E$. We need both variants, (directed and undirected), and freely use whichever definition is more suitable in a particular situation. As in this thesis the graph class in question is always evident from the context, we abusively denote by $\mathcal{G}$ the class of graphs in that very category. Given a graph $G \in \mathcal{G}$, we denote by $V(G)$ and $E(G)$ its vertex set and edge set respectively.

By $n = |G| = |V(G)|$ we denote the number of vertices of a graph, its *size*. By m we denote the number of edges $|E(G)|$ in the graph. A simple graph in which every pair of distinct vertices forms an edge is *complete*. For a subset of vertices $U \subseteq V(G)$ the *induced subgraph* on U is the graph $G[U] := (U, E(G) \cap \{\{u, u'\} \mid u, u' \in U\})$, i.e., the graph that consists of the vertices from U which share an edge, if they do so in G. Conversely, we define $G - U$ as the graph obtained by *deleting vertices* U from V, i.e., as the graph $G[V \setminus U]$.

When dealing with graphs in the context of graph isomorphisms, it is convenient to work with colored graphs:

Definition 2 (colored graph). A *vertex colored graph* is a graph $G = (V, E)$ together with a map $c_G \colon V \to M$ from the vertices into some set of colors M.

An *edge colored graph* is a graph $G = (V, E)$ together with a map $c_G \colon E \to M$ from the edges into some set of colors M.

For a colored graph G, of either type, we denote by c_G its color map, and consider the triple (V, E, c_G) as a the colored graph itself. Whether we consider an edge or a vertex coloring is implied by the context, in which we use the colored graphs.

Babai's chapter on automorphism groups, isomorphism and reconstruction in the Handbook of Combinatorics [4] is a good starting point to get an overview of the field of these concepts. We continue with the definition of an isomorphism:

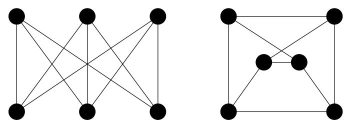
Figure 2.1: Isomorphic graphs

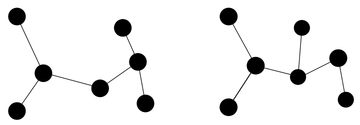
Figure 2.2: Non-isomorphic graphs

Definition 3 (graph isomorphism). Given graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ a *graph isomorphism* from G_1 to G_2 is bijection $\phi\colon V_1 \to V_2$ such that for all $v, v' \in V_1$ we have $\{v, v'\} \in E_1$ if and only if $\{\phi(v), \phi(v')\} \in E_2$.

Thus an isomorphism is a bijection on the vertices that preserves adjacency and non-adjacency. Figures 2.1 and 2.2 depict a pair of isomorphic and a pair of non-isomorphic graphs. For two graphs G_1, G_2 we write $G_1 \cong G_2$ (respectively $G_1 \ncong G_2$), if the graphs are isomorphic (respectively non-isomorphic). The *isomorphism type* of a graph G is the class of graphs that are isomorphic to G. An *automorphism* of a graph G is, as for any category, an isomorphism from G to itself. For any graph G, the set of automorphisms forms a group, *the automorphism group* of G, which we denote by $\mathrm{Aut}(G)$.

When we work with colored graphs, we impose on the bijection the restriction to preserve the colors:

Definition 4 (graph isomorphism of colored graphs). An isomorphism from a vertex colored graph G_1 to a vertex colored graph G_2 is an (uncolored) isomorphism ϕ from G_1 to G_2 such that additionally for all $v \in V(G_1)$ we have $c_{G_1}(v) = c_{G_2}(\phi(v))$.

Analogously, whenever we consider edge colored graphs, we require that for all edges $\{v, v'\} \in E(G)$ we have $c_{G_1}(\{v, v'\}) = c_{G_2}(\{\phi(v), \phi(v')\})$.

This chapter of the thesis is mainly concerned with the corresponding computational problem:

Problem 1 (graph isomorphism problem). Given two finite graphs G_1, G_2, the *graph isomorphism problem* (GI) is the task to decide whether G_1 and G_2 are isomorphic.

As our central question is a computational problem, for the remainder of this chapter we assume that all graphs are finite.

Basic knowledge in permutation group theory is indispensable when dealing with the graph isomorphism problem. In particular we later require the concept of orbit partitions.

Definition 5 (orbit). Given a graph G and a vertex $v \in V(G)$ the *orbit of* v is the set of images of v under all automorphisms of G.

For colored graphs the automorphisms in this definition are those preserving colors. The relation $v_1 \sim v_2$ that holds for vertices $v_1, v_2 \in V(G)$ if v_2 is in the orbit of v_1 is an equivalence relation. Its equivalence classes, the orbits, form a partition of the vertices, the aforementioned *orbit partition*.

We frequently use the adjective *invariant* to describe a function that is invariant under graph isomorphisms, i.e., evaluates equally for isomorphic graphs. Here the invariance must hold under the type of isomorphisms that are under consideration in the respective context (e.g., it must respect color constraints, if isomorphisms of colored graphs are considered).

A *complete invariant* is a function that does not only map isomorphic graphs to the same value but also maps non-isomorphic graphs to different values. A complete invariant, that is computable within a certain time bound, solves the graph isomorphism problem within this time bound (apart form the additional time needed to compare values of the invariant). A canonical labeling is a special type of complete invariant:

Definition 6 (canonical labeling). A *canonical labeling* is a complete invariant that maps graphs on n vertices to isomorphic graphs on the vertex set $\{1, \ldots, n\}$.

Thus, with a *canonical labeling*, for every n-vertex graph G we obtain a function χ_G, that assigns the labels $1, \ldots, n$ to the vertices of G. Furthermore, the map that is given by $\chi_G^{-1}(i) \mapsto \chi_{G'}^{-1}(i)$ is an isomorphism for any pair (G, G') of isomorphic graphs with n vertices. In other words, the isomorphism is formed by mapping vertices in G to those with equal label in G'. Computationally, this implies that as soon as we know the canonical labelings of two graphs, we can trivially check whether the graphs are isomorphic. Moreover, if they are isomorphic, we obtain an isomorphism. Brendan McKay's graph isomorphism solver Nauty uses this approach (see the Nauty user guide [88] and Section 2.2).

The theoretical complexity of the graph isomorphism problem is still unknown. The problem has properties that are presumably not shared by other $\mathcal{NP}$-hard problems: Goldreich, Micali and Wigderson [47] showed that GI has an interactive proof system and as Schöning [115] shows GI is low in the hierarchy. For an overview of known results on the complexity of graph isomorphism see [72]. This implies that the polynomial hierarchy collapses if GI is $\mathcal{NP}$-complete.

Many isomorphism questions are equally hard as graph isomorphism. We therefore introduce a complexity class that contains these problems:

Definition 7 (graph isomorphism-complete). A problem P is GI-*complete* if there is a polynomial-time reduction from P to GI and vice versa.

We now discuss the relation of this complexity to other classes and its prominent members:

2.1.1 Reductions: equivalent and non-equivalent problems

When dealing with problems "similar" to GI, it appears that many of those fall into three classes: they are $\mathcal{NP}$-complete, they are GI-complete or they are polynomial-time solvable. Since the graph isomorphism problem ranges between problems in $\mathcal{P}$

and $\mathcal{NP}$-complete problems, we exclusively look at polynomial-time reductions, as opposed to logarithmic space reductions.

$\mathcal{NP}$-complete variants

The most prominent example of an $\mathcal{NP}$-complete problem in this area is presumably the subgraph isomorphism problem. It takes various forms: Given two graphs G_1 and G_2 one asks whether G_1 is a subgraph of G_2, induced or not induced, or one tries to determine the largest subgraph common to G_1 and G_2. All three variants are $\mathcal{NP}$-complete which can easily be seen by a reduction of Max-Clique to the special case in which G_1 is a complete graph. Being a generalization of graph coloring, the question whether there exists a homomorphism from G_1 to G_2 is also $\mathcal{NP}$-complete. As Lubiw [85] shows, the problem that asks whether there exists an automorphism of a graph which does not fix any vertex is $\mathcal{NP}$-complete, which stands out from other variants that are Gi-complete.

Graph isomorphism-complete variants

Many problems concerning structural equivalence are easily seen to be Gi-complete. The general problem of hypergraph isomorphism is Gi-complete and with it the isomorphism problem of simplicial complexes. This stays true if colored hypergraph isomorphisms are considered. More generally, the isomorphism problem for general relational structures is Gi-complete, as shown by Miller [98]. Mathon [87] shows that the problem of counting the number of isomorphisms between two graphs is Gi-complete. The proof of Theorem 1, given below, shows a color reduction method, which shows that the problems of colored and uncolored graph isomorphism are equivalent. The natural graph classes given by any choice of loops or loopless, multiple edges, bipartite, connected, colored or regular as properties also have Gi-complete isomorphism problems. We therefore refer to Gi-complete problems simply as isomorphism-complete.

Basin [8] shows that a certain term equality problem, of terms also containing commutative variable-binding operators, is isomorphism-complete. Colbourn and Colbourn [27] show that deciding isomorphism of block designs is Gi-complete.

Furthermore, deciding isomorphism of finite semigroups (given by multiplication table) and finite automata (Booth [15]), of finitely represented algebras (Kozen [75]) and of convex polytopes (Kaibel and Schwartz [66]) is Gi-complete as well. There are also classes of groups (as shown for example by Garzon and Zalcstein [46]) for which deciding isomorphism, when the groups are given via presentations, is Gi-complete. In accordance with this Droms [34] shows that right-angled Artin groups are isomorphic if and only if their underlying graphs are isomorphic.

There are only a limited number of equivalence results for problems which are not directly related to the isomorphism of combinatorial structures. For example, Kutz [78] shows that deciding if a subdivision digraph (a digraph, in which every edge has been subdivided) with positive minimal in- and outdegree has a k-th root, is isomorphism-complete. Kozen [76] shows that finding a clique of a certain size in M-graphs is

Gɪ-complete. Feigenbaum and Schäffer [40] show that the question whether a graph decomposes non-trivially as a lexicographic product is Gɪ-complete. Hemaspaandra, Hemaspaandra, Radziszowski and Tripathi [60] show that various graph reconstruction problems are Gɪ-complete.

When we relax the allowed reduction method from many-one reduction to any form of Turing-reduction, we obtain even more Gɪ-complete problems. Under these, Gɪ is set apart from the typical $\mathcal{NP}$-complete problem by the fact that it is equivalent to its counting version: the task of determining the number of isomorphisms between two graphs. This also renders the problem of determining the size of the automorphism group of a graph Gɪ-complete. Moreover computing generators for the automorphism group is Gɪ-complete as well.

Polynomial variants

When restricting the class of graphs in a severe way, i.e., in a way such that we cannot show Gɪ-completeness anymore, we can expect that the restricted problem falls into $\mathcal{P}$. For several restricted classes of graphs, polynomial-time algorithms are known. The most prominent are planar graphs [125, 62], minor closed graphs [107], graphs with bounded eigenvalue multiplicity [6], graphs of bounded genus [41, 84, 100], graphs of bounded degree [86], graphs of bounded color class size [44] and graphs of bounded treewidth [13]. Isomorphism of random graphs [5] can be tested in expected polynomial time.

Other variants

Being able to compute a canonical labeling for all graphs certainly suffices to decide graph isomorphism. Conversely though, it is not known whether computing a canonical labeling is computationally harder than Gɪ.

Group isomorphism (when finite groups are given by their multiplication table) is reducible to graph isomorphism as shown by Miller and Monk [98]. The converse is again unknown. Tarjan showed that group isomorphism can be solved in $\mathcal{O}(n^{\log(n)+\mathcal{O}(1)})$, thus showing a reduction of graph isomorphism to it would improve the best known running time for the Gɪ problem. (Apparently Tarjan never published this algorithm; however, the algorithm and its running time bound can be found in [99]).

Though deciding isomorphism for block designs is Gɪ-complete [27], for the "empirically difficult small cases", the projective planes and Hadamard matrices, Gɪ-completeness is not known (see Section 2.8). Miller [99] provides an algorithm with which isomorphism of projective planes can be decided in $n^{\mathcal{O}(\log\log(n))}$. Leon [82] provides an algorithm that computes the automorphism group of a Hadamard matrix, which can also be used to decide equivalence of Hadamard matrices in $n^{\mathcal{O}(\log n)}$.

Explicit reductions

The fact that Gɪ with colored graphs reduces to Gɪ of uncolored graphs is folklore. There are many polynomial-time reductions; we provide one for completeness. This particular reduction increases genus, treewidth and maximum degree by at most a constant:

Theorem 1 (reduction of colored to uncolored graph isomorphism). *The graph isomorphism problem for colored graphs polynomial-time reduces to the uncolored graph isomorphism problem.*

Proof. Assume we are given two colored graphs G_1, G_2 on n vertices. If the sets of colors used for the graphs are not equal, we reduce the problem to a **No**-instance of uncolored graph isomorphism, i.e., some fixed pair of non-isomorphic graphs.

If the graphs use the same color set, we attach to every vertex a rooted tree whose isomorphism type is in one-to-one correspondence with the color of the vertex: We choose a canonical bijection of the color set to the set of rooted trees for which every leaf is at height $\lceil \log_2(n) \rceil$ and which has a maximum degree of 3. Such a bijection is given by the following method: We number the colors with integers in $\{0, \ldots, n-1\}$. To a color with binary encoding $a_0 a_1 \ldots a_{\lceil \log_2 n - 1 \rceil}$ we assign the tree for which every vertex on height i has exactly $a_i + 1$ children.

We obtain two new graphs G'_1, G'_2 of size at most $\mathcal{O}(n^2)$. By induction on the height, it is easy to show that these new graphs are isomorphic if and only if the original graphs are. $\square$

Note that this reduction reduces trees to trees and does not increase the maximum degree by more than 3 (if we choose the dummy **No**-instance wisely.) The reduction shown may square the number of vertices in order to reduce the colors. Reductions to smaller size graphs are possible as well, e.g., by attaching different subgraphs not contained in the original graphs to encode colors.

Overall there are numerous color reductions. Possibly less known is the degree reduction method of Zemlyachenko [3].

Theorem 2 (degree reduction of graph isomorphism [Zemlyachenko [3] (1981)]). *There exists a reduction that, given two graphs of size n and of maximal degree at most d, produces two graphs of maximal degree at most $\lceil d/2 \rceil$. These graphs are isomorphic if and only if the original graphs are. The new graphs are of size $\mathcal{O}(n^{2\frac{n}{d}})$ and they may be computed in time polynomial in that size.*

This degree reduction has been used to obtain the moderately exponential graph isomorphism algorithm [3]. We do not expect to find a trivial way to reduce the degree of the graph. The reason for this is the following fact: For $d > 4$ there is no colored graph of maximum degree $d' < d$ which has an orbit of size d on which the induced group operation is the full permutation group. Miller [98] calls such a (non-existent) graph concisely a d-gadget. If they existed, we could reduce the degree of a graph by replacing every vertex of degree d with such a d-gadget. (The fact that these graphs do not exist can be seen via the composition series of the symmetric group S_n, as for $n > 4$ this composition series contains simple groups that are not contained in S_{n-1}.)

2.2 Brendan McKay's Nauty

We use this section to review the Nauty algorithm designed by McKay [92]. To do so we introduce the necessary vocabulary. Most of these definitions are taken from the Nauty user guide [89]. Nauty is an algorithm that produces a canonical labeling of a given input graph. Isomorphism of two graphs can then easily be checked via equivalence of the respective canonical labelings.

A *partition* of G is a partition of the vertices of G. When Nauty performs operations on such partitions it maintains and takes into account an ordering of the partition classes. The partition classes are also called *cells*. A vertex coloring of a graph induces a partition of the vertices as the preimages of colors. A cell of size one in a partition is called *singleton*. A partition that contains only singletons is *discrete*. A partition π is called *finer* than a partition π' if all cells of π are subsets of cells of π'. Under these conditions π' is *coarser* than π. The relation "finer than" defines a partial order on the partitions of a vertex set. (Moreover it defines a lattice on the partitions, a fact we will not use). Recall that the orbit partition of a graph is the partition given by the sets of orbits of the automorphism group.

Definition 8 (refinement). A *refinement* is a function invariant under graph isomorphism that maps every colored graph $G = (V, E, c)$ to a colored graph $G' = (V, E, c')$, such that the induced partition of c' is finer than the induced partition of c.

More precisely a refinement is a functor from the category of finite colored graphs (where its morphisms are the isomorphisms) to itself. *Invariance under graph isomorphisms* means that for any isomorphic copy G' of G, the refinement colors corresponding vertices in G and G' with the same color. In particular, this causes any two vertices $v, v' \in G$ which lie in the same orbit (under the color respecting automorphism group of G) to have images that lie in the same orbit of G'. Thus a refined coloring of a graph induces a partition that is finer than the partition induced by the original coloring and coarser than the orbit partition of the graph. In other words, orbits will never be split up.

Definition 9 (vertex invariant). A *vertex invariant* is a function that maps the vertices of a colored graph to some set M and is invariant under graph isomorphism.

Any vertex invariant can be used to refine a partition by differentiating vertices according to their image under the invariant.

Definition 10 (stable partition). The *stable partition* under a refinement r is the finest partition obtained by any number of repeated refinement steps performed with r.

Any partition of a set of size n can be refined at most n times before it becomes discrete. Therefore any refinement repeated sufficiently often has to stabilize and the stable partition under a given refinement is well defined. (Recall that for our computational problem we require the graphs to be finite). A particular and basic refinement is the one that assigns to each vertex a color that depends on the number of neighbors of that vertex and the neighbors' colors:

Definition 11 (naïve vertex refinement). The *naïve vertex refinement* maps every colored graph $G = (V, E, c)$ with $c \colon V \to M = \{m_1, m_2, \ldots, m_{|M|}\}$ to a new colored graph $G' = (V, E, c')$ with $c' \colon V \to M \times \mathbb{N}^M$ given by

$$c'(v) := \big(c(v), |\{v' \mid \{v, v'\} \in E, c(v') = m_1\}|, \ldots, |\{v' \mid \{v, v'\} \in E, c(v') = m_{|M|}\}|\big) .$$

The newly assigned color of a vertex v thus is the tuple consisting of the previous color of v, the number of neighbors in color m_1, the number of neighbors in color m_2, and so on. This refinement is also called the 1-dimensional Weisfeiler-Lehman refinement (for higher dimensions see Definition 13).

A partition is called *equitable* if it is stable under the naïve vertex refinement.

Definition 12 (individualization). Given a colored graph $G = (V, E, c)$, an *individualization* of the vertex u of G is a colored graph $G_u := (V, E, c_u)$ where

$$c_u(v) := \begin{cases} c(v) & \text{if } v \neq u, \\ m' & \text{if } v = u, \end{cases}$$

where m' is a new color that is not in the image of c.

In an individualization, the individualized vertex thus gets a color that distinguishes it from the other vertices. Given a graph as input, Nauty uses the following backtracking procedure: It constructs a search tree, in which every node corresponds to a partition of the input graphs. (To avoid confusion with the vertices of the graph, we call the vertices of the search tree nodes.) The root of the search tree corresponds to the naïvely vertex refined coloring of the input. (We choose a uniform coloring for uncolored inputs.) Then, for each node in the search tree, which does not correspond to a discrete partition, Nauty recursively picks a *target cell*, i.e., a partition class, of this coloring according to some heuristic rule. One by one each vertex in this cell is individualized and the resulting coloring is naïvely vertex refined. The obtained coloring is a child of the current node. The search then proceeds, in a depth first search manner, with the children. The leafs of the search are all associated with discrete partitions. According to a deterministic rule, one of the leafs is taken as a canonical labeling of the graph. (The deterministic rule lexicographically orders the leafs by the node types on the path from the root to the leaf, and chooses the leaf minimal in this order.) The procedure we have described so far already suffices to correctly determine whether two graphs are isomorphic, but would demand infeasible exponential computation time, even when a complete graph is given as input. Nauty therefore uses two methods to prune the search tree. First, since the automorphism group acts on the search tree, detected automorphisms can reveal the equivalence of search nodes. If for some search tree node Nauty detects that two vertices in the target cell lie in the same orbit (of the automorphism group of the colored graph corresponding to the current node), only one of the vertices has to be individualized, as the other one yields an equivalent branch of the search tree with equivalent leafs. Second, Nauty uses an indicator function (and in particular the lexicographic ordering of the leafs) to determine ahead of time that some nodes do not have to be individualized. This concludes a very rough sketch of the algorithm, which omitted crucial details necessary to obtain a desired efficiency.

There are several other algorithms that use the individualization refinement technique. Among those are Saucy [31], which is an algorithm that exploits sparsity of input graphs, Bliss [65], which was derived by using efficient data structures and algorithm engineering, and Traces [106], which uses specific individualization and refinement rules to drastically decrease the number of nodes visited in the search tree.

We now present the Weisfeiler-Lehman method, our second example of a graph isomorphism algorithm.

2.3 The Weisfeiler-Lehman method

The Weisfeiler-Lehman method is a powerful refinement that uses a k-vertex tuple coloring procedure. The 1-dimensional Weisfeiler-Lehman refinement was introduced in Definition 11. Its k-dimensional generalization colors k-tuples by considering the way they are embedded in the graph.

Definition 13 (k-dimensional Weisfeiler-Lehman coloring procedure). Let $k \geq 2$ be natural number and G a colored graph. For every k-tuple of (not necessarily distinct) vertices $(v_1, \ldots, v_k)$ define $\mathrm{wl}_0^k(v_1, \ldots, v_k)$ as the isomorphism type of the colored subgraph induced by $(v_1, \ldots, v_k)$. (Here we take the order of the vertices into account.) I.e., $\mathrm{wl}_0(v_1, \ldots, v_k) = \mathrm{wl}_0(v'_1, \ldots, v'_k)$ if and only if the map that sends v_j to v'_j for $j \in \{1, \ldots, k\}$ is an isomorphism of the induced colored subgraph on the vertices $(v_1, \ldots, v_k)$ respectively $(v'_1, \ldots, v'_k)$. Iteratively for $i \geq 0$ we define $\mathrm{wl}_{i+1}^k(v_1, \ldots, v_k) := \big(\mathrm{wl}_i^k(v_1, \ldots, v_k), \mathcal{M}_i^k\big)$ where $\mathcal{M}_i^k$ is the multiset given by $\mathcal{M}_i^k :=$

$$\{\{(\mathrm{wl}_i^k(w, v_2, \ldots, v_k), \mathrm{wl}_i^k(v_1, w, v_3, \ldots, v_k), \ldots, \mathrm{wl}_i^k(v_1, \ldots, v_{k-1}, w)) \mid w \in V\}\}.$$

The colors wl_i^k are the *colors obtained in the i-th iteration of the Weisfeiler-Lehman coloring procedure.*

Thus in every iteration of the Weisfeiler-Lehman coloring procedure, every tuple $(v_1, \ldots, v_k)$ is given a new color. This new color consists of the previous color of the respective tuple and the multiset obtained by substituting successively each v_i by w for all vertices w in the graph.

Observe that the k-dimensional Weisfeiler-Lehman coloring procedure refinement is invariant under graph isomorphism. As for the 1-dimensional case, where only the vertices (i.e., 1-tuples) are colored, this procedure stabilizes. By abuse of notation we define $\mathrm{wl}_\infty^k(v_1, v_2, \ldots, v_k)$ to be this stable coloring. (The coloring continues to change, but the induced partition of the set of k-tuples does not. One way to remedy this is to define $\mathrm{wl}_\infty^k(v_1, v_2, \ldots, v_k)$ as $\mathrm{wl}_i^k(v_1, v_2, \ldots, v_k)$ where i is the least positive integer such that the induced partition in step i is equivalent to the induced partition in step $i + 1$.)

Using the stable coloring wl_∞^k, we use the k-tuple coloring procedure to produce the k-dimensional Weisfeiler-Lehman vertex refinement, a refinement in the sense of

Definition 8, that colors vertices, as opposed to tuples of vertices. To color a vertex v, we use the color of the tuple that consists only of the vertex v:

Definition 14 (k-dimensional Weisfeiler-Lehman vertex refinement). Given a colored graph $G = (V, E, c)$, define $G' = (V, E, c')$ as the *k-dimensional Weisfeiler-Lehman vertex refinement*, where $c'(v) = \mathrm{wl}^k_\infty(v, v, \ldots, v)$.

Intuitively this refinement is finer for larger k, since more colors and further information is used to differentiate the tuples of vertices. The partition induced by the k-dimensional Weisfeiler-Lehman vertex refinement is stable, thus in the 1-dimensional case, the corresponding partition is obtained by the stable partition of the naïve vertex refinement.

With brute force, a single iteration of the k-dimensional Weisfeiler-Lehman coloring procedure can be computed in $\mathcal{O}(kn^{k+1})$ time. Immerman and Lander [63] show that a stable refinement can be computed in $\mathcal{O}(k^2 n^{k+1} \log n)$. Even though it may be possible to improve this bound via fast matrix multiplication, we expect a lower bound of $\Omega(n^k)$, as there are n^k tuples that must obtain a color.

The *k-dimensional Weisfeiler-Lehman algorithm*, corresponding to the just-defined vertex refinement, performs the refinement on two input graphs. It then claims that the graphs are isomorphic if the colors with their multiplicity are equal in both graphs. However, we later see with Theorem 5 that for any k this algorithm has false positives: graphs, which are not distinguished by their color refinement, but which are not isomorphic. In other words, for any fixed k the k-dimensional Weisfeiler-Lehman algorithm solves graph isomorphism only for a subclass of graphs.

The Weisfeiler-Lehman algorithm subsumes almost all combinatorial graph algorithms that are not based on the group theoretic method, (see Section 2.5). An exception to this might be the problem of deciding isomorphism of graphs of bounded eigenvalue multiplicity, for which Fürer gave a combinatorial algorithm [43]. To demonstrate the power of the Weisfeiler-Lehman method, we cite two theorems that handle graph isomorphism for two natural graph classes:

Theorem 3 (k-dimensional Weisfeiler-Lehman algorithm solves bounded genus [Grohe [55] (2000)]). *For any genus bound g there is a number $f(g)$ such that the $f(g)$-dimensional Weisfeiler-Lehman algorithm solves* Gi *for graphs with a genus of at most g.*

Prior to the proof of this theorem, Grohe and Mariño showed that the same is true when the treewidth is taken as parameter:

Theorem 4 (k-dimensional Weisfeiler-Lehman algorithm solves bounded treewidth [Grohe, Mariño [56] (1999)]). *For any treewidth bound w there is a number $f(w)$ such that the $f(w)$-dimensional Weisfeiler-Lehman algorithm solves* Gi *for graphs with a treewidth of at most w.*

Before we explore another graph isomorphism algorithm, namely Luks' algorithm that solves Gi for graphs of bounded degree, we first investigate families of graphs of bounded degree, which the k-dimensional Weisfeiler-Lehman algorithm fails to differentiate and for which Nauty fails to yield polynomial running time.

2.4 The Cai-Fürer-Immerman construction and Miyazaki graphs

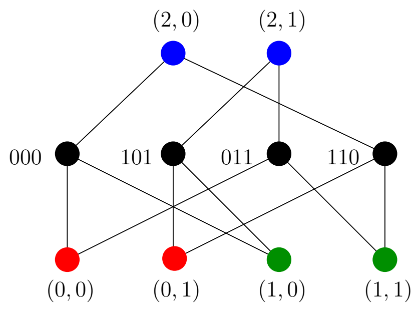

Figure 2.3: The Figure depicts the Fürer gadget F_3. The 4 middle vertices are shown in the middle row (with color 0 depicted as black). Three pairs of equally colored outer vertices are shown above and below the middle vertices, (the colors 1, 2 and 3 are shown in red, green and blue respectively).

In this section we outline the Cai-Fürer-Immerman construction. It produces pairs of graphs that are difficult for various approaches to the graph isomorphism problem. Cai, Fürer and Immerman [23] show that for any k the k-dimensional Weisfeiler-Lehman algorithm cannot distinguish all graphs, not even those of bounded degree. Using their construction, Miyazaki [102] shows that Nauty has exponential running time on a family of graphs of bounded degree. To explain the construction we first need to define the Fürer gadgets [42] F_i. (See Figure 2.3 for the graph F_3.)

Definition 15 (Fürer gadget). For any non-negative integer k we define the *Fürer gadget* $F_k = (V, E, c)$ as the graph on the vertex set $V := O_k \cup M_k$, where $O_k := \{1, \dots, k\} \times \{0, 1\}$, and M_k is the set of 0-1-strings of length k with an even number of entries equal to 1, i.e.,

$$M_k := \{\sigma_1 \dots \sigma_k \in \{0,1\}^k \mid |\{\sigma_i \neq 1\}| \text{ is even}\}.$$

The edge set is given by

$$E := \big\{\{(i, j), \sigma_1 \dots \sigma_k\} \mid i \in \{1, \dots, k\}, j \in \{0, 1\} \wedge \sigma_i = j\big\}.$$

The map $c \colon V \to \{0, \dots, k\}$ colors the vertices $(i, j) \in O_k$, with $i \in \{1, \dots, k\}$ and $j \in \{0, 1\}$, such that $c((i, j)) = i$. All remaining vertices, i.e., those in M_k, are colored with color 0.

Thus the Fürer gadget F_k contains a set of *middle* vertices M_k, and each of them corresponds to a 0-1-sequences of length k. For every index $i \in \{1, \dots, k\}$ it also contains two *outer* vertices $(i, 0), (i, 1) \in O_k$. For $i \in \{1, \dots, k\}$ outer vertex $(i, 0)$ (respectively $(i, 1)$) is joined to all middle vertices that correspond to a sequence with entry 0 (respectively 1) at position i. Each set $\{(i,0),(i,1)\}$ of outer vertices forms a color class. The middle vertices also form a color class.

The automorphism group of the colored graph F_k is isomorphic to $\mathbb{Z}_2^{k-1}$, the $(k-2)$-fold direct product of cyclic groups of order 2. This automorphism group acts on the pairs of equally colored outer vertices, i.e. on the sets $\{(i,0)(i,1)\}$ with $i \in \{1,\dots,k\}$. Any automorphism transposes an even number of these pairs. Conversely any permutation of the outer vertices that transposes an even number of these pairs can be extended to an automorphism of the whole graph. This action is faithful, i.e., only the trivial automorphism fixes all outer vertices. The graph F_k has $2^{k-1}+2k$ vertices and maximum degree of $\max\{k, 2^{k-2}\}$.

The Fürer gadgets may be used as a building block to construct difficult graph isomorphism instances. To do so, we replace in a base graph G every vertex by a Fürer gadget. The edges in the graph G determine how the vertices from different replacement gadgets are connected with extra edges. We now explain this construction in detail (the middle graph of Figure 2.4 depicts an example of the construction):

Definition 16 (replacement with Fürer gadgets). Given a base graph G, we define $\mathrm{CFI}(G)$, the *replacement with Fürer gadgets*, as the graph obtained by replacing each vertex of G with a Fürer gadget of specific size: First, for every vertex $v \in V(G)$ we replace v with the graph $F_{\deg(v)}$ where $\deg(v)$ is the degree of v. (We index the colors of this replaced graph by the index v, such that the sets of colors used in replacements for different vertices v and v' from G are disjoint.) Second, we associate with every edge e in G incident to v one pair of outer vertices of equal color. We denote this pair by (a_e^v, b_e^v). Every edge $e = \{v, v'\}$ in the original graph is then associated with two such pairs in $\mathrm{CFI}(G)$: one pair (a_e^v, b_e^v) in the replacement graph of v and one pair $(a_e^{v'}, b_e^{v'})$ in the replacement graph of v'. Besides edges within the gadgets, for every edge $e = \{v, v'\}$ in G, we also add the edges joining a_e^v with $a_e^{v'}$ and b_e^v with $b_e^{v'}$ to the new graph $\mathrm{CFI}(G)$.

The graph that we obtain by replacement with Fürer gadgets has two type of edges: It contains edges that are completely contained in one of the Fürer gadgets. We call these edges *internal*. And it contains edges that connect different Fürer gadgets. We call these edges *external*. External edges appear in pairs.

For connected graphs, additionally to this replacement, we define the twisted replacement to be the same replacement graph, apart from one pair of external edges, which is twisted:

Definition 17 (twisted replacement with Fürer gadgets). For every connected non-trivial graph G we define the *twisted replacement with Fürer gadgets* $\widetilde{\mathrm{CFI}}(G)$, as the graph obtained with the untwisted replacement procedure (Definition 16), except for exactly one edge $e = \{v, v'\}$, associated to (a_e^v, b_e^v) and $(a_e^{v'}, b_e^{v'})$. For this edge we insert the edges $\{a_e^v, b_e^{v'}\}$ and $\{b_e^v, a_e^{v'}\}$, instead of the untwisted pair of edges, (i.e, instead of the two edges $\{a_e^v, a_e^{v'}\}$ and $\{b_e^v, b_e^{v'}\}$).

Figure 2.4 shows the replacement and the twist operation of an example graph. The automorphism group of a graph $\mathrm{CFI}(G)$ is the elementary Abelian 2-group of rank

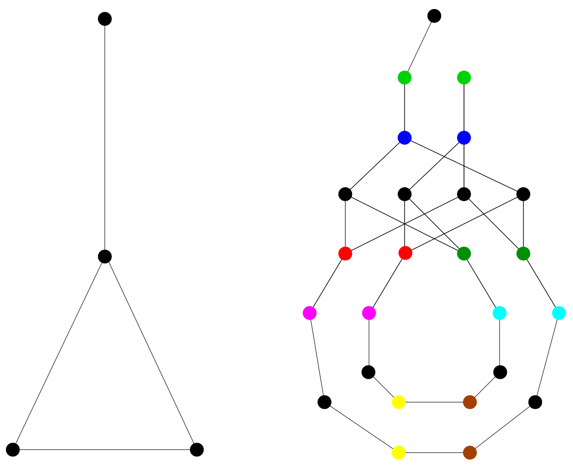
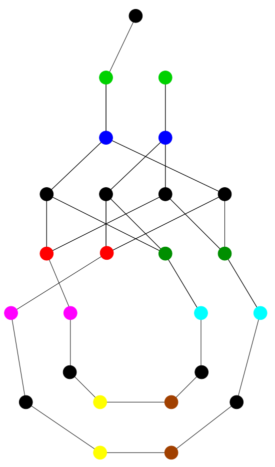

Figure 2.4: The Figure shows a base graph (left), its replacement with Fürer gadgets (middle) and the corresponding twisted replacement (right). The vertex of degree 3 in the base graph has been replaced with the graph F_3, shown in Figure 2.3. The twist is introduced at the pair of edges associated with the edge in the base graph that connects the vertex of degree 3 and the vertex of degree 2 in the lower left corner. All middle vertices are shown in black. The outer vertices from different replacement gadgets have been given different colors.

equal to the dimension of the cycle space of G. The automorphism group of $\widetilde{\mathrm{CFI}}(G)$ is isomorphic to the one of $\mathrm{CFI}(G)$.

Observe that the twisted replacement of a base graph G is well defined up to isomorphism: Since the original graph is required to be connected, it suffices to show that for two incident edges e and e' in the base graph G, the two graphs obtained by twisting one of the corresponding pairs of external edges in $\mathrm{CFI}(G)$ are isomorphic. Assume e and e' are incident in v. Let (a^v_e, b^v_e) and $(a^v_{e'}, b^v_{e'})$ be the pairs in the replacement of v associated to e and e' respectively, then by construction there an automorphism Fürer gadget used to replace v, that interchanges a^v_e with b^v_e and $a^v_{e'}$ with $b^v_{e'}$, and leaves all other outer vertices fixed. In other words, the graph F_k has been designed such that the twist can be moved among pairs of external edges that originate from edges incident in the base graph G, with the help of an automorphism of F_k. Contrarily, the graphs $\mathrm{CFI}(G)$ and $\widetilde{\mathrm{CFI}}(G)$ are not isomorphic. Since any automorphism of $\mathrm{CFI}(G)$ transposes an even number of pairs (a, b), the parity of the number of twists (pairs (a^v_e, b^v_e) and $(a^{v'}_e, b^{v'}_e)$ where a^v_e is adjacent to $b^{v'}_e$ and b^v_e is adjacent to $a^{v'}_e$) is a graph isomorphism invariant. For $\mathrm{CFI}(G)$ and $\widetilde{\mathrm{CFI}}(G)$ this parity is 0 and 1 respectively. The CFI-construction, i.e., the application of the replacement with Fürer gadgets and the twisted replacement with Fürer gadgets to a connected base graph G, thus yields two non-isomorphic graphs.

With this, we next describe a class of graphs which the k-dimensional Weisfeiler-Lehman algorithm cannot distinguish. We first recall the notion of a balanced vertex separator:

Definition 18 (balanced vertex separator). A *balanced vertex separator* of a graph G is a subset of its vertices $S \subseteq V(G)$, such that no component of $G - S$ has more than $|V(G)|/2$ vertices.

It turns out that if the CFI-construction is applied to graphs without small balanced separators, it is very difficult to determine whether a twist has been introduced. Intuitively this is due to the fact that the twist can move around the graph. This movement cannot easily be prohibited by individualizations of vertices, as one has to individualize every vertex within some separator. If separators are not small, many individualizations are required. In their groundbreaking paper Cai, Fürer and Immerman develop this intuition and turn it into a formal argument:

Theorem 5 (criterion for indistinguishability of graphs obtained with the CFI-construction [Cai, Fürer, Immerman [23] (1992)]). *Let G be a graph with no balanced vertex separator smaller than $k+1$. Then $\mathrm{CFI}(G)$ and $\widetilde{\mathrm{CFI}}(G)$ cannot be distinguished by the k-dimensional Weisfeiler-Lehman algorithm.*

With this theorem at hand we may now construct a family of graphs (even of bounded degree) which for any fixed k cannot be distinguished by the k-dimensional Weisfeiler-Lehman algorithm. One performs the CFI-construction to a family of bounded degree expanders. As expanders, they cannot contain small balanced vertex separators:

Corollary 1 (graphs indistinguishable for the Weisfeiler-Lehman algorithm [Cai, Fürer, Immerman [23] (1992)]). *There is a family $\{(G_i, G'_i) \mid i \in \mathbb{N}\}$ of pairs of non-isomorphic regular graphs of degree 3 and color class size bounded by 4 with $\mathcal{O}(i)$ vertices such that for any k the k-dimensional Weisfeiler-Lehman algorithm cannot distinguish between the graphs G_i and G'_i for any $i \geq k$.*

Miyazaki [102] used the CFI-construction to show that there is a family of graphs for which Nauty has exponential running time. In particular he applied it to the 3-regular multigraphs obtained by the following definition: For $k \in \mathbb{N}$ define M_k the *Miyazaki graph* as the graph on the vertex set $V(M_k) := \{v_1, \ldots, v_k, w_1, \ldots, w_k\}$ with edge multiset $E(M_k) :=$

$$\{\!\{\, \{v_1, v_1\}, \{w_k, w_k\}, \{v_i, w_i \mid i \in \{1, \ldots, k\}\}, 2 \cdot \{w_i, v_{i+1} \mid i \in \{1, \ldots, k-1\}\} \,\}\!\}.$$

I.e., in this multiset the edges $\{w_i, v_{i+1}\}$ appear twice. Figure 2.5 shows the Miyazaki graph M_3. We observe that, if the CFI-construction is applied to M_k, the multiedges and the loops are assigned to different endpoints, thus $\mathrm{CFI}(M_k)$ is a simple graph (one without multiedges). With slight ambiguity, we call the graphs M_k as well as the graphs $\mathrm{CFI}(M_k)$ Miyazaki graphs.

With the help of these graphs, one may force Nauty to have exponential running time:

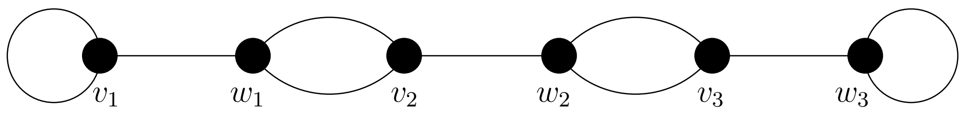

Figure 2.5: The Miyazaki graph M_3

Theorem 6 (exponential running time of Nauty [Miyazaki [102] (1995)]). *There is an ordering of the colors for the family of graphs* CFI(M_k) *such that Nauty has exponential running time for these graphs.*

We revisit the CFI-construction in Subsection 2.10.3. We now return to the discussion of graph isomorphism algorithms, and consider Luks' algorithm for graphs of bounded degree.

2.5 Eugene Luks' bounded degree algorithm

In 1982 Luks [86] designed a graph isomorphism algorithm that runs in polynomial time for graphs with bounded degree. It is, opposed to the combinatorial Weisfeiler-Lehman method, of group theoretic nature. Luks reduced GI to orbit classification of permutation groups, whose composition factors are subgroups of S_n, the symmetric group on n elements. Together with Zemlyachenko's degree reduction (Theorem 2), Babai [3] obtained an $e^{\mathcal{O}\left(\sqrt{n\log(n)}\right)}$ deterministic algorithm for GI in general.

Theorem 7 (polynomial time isomorphism algorithm for graphs of bounded degree [Luks [86](1982)]). *There is a polynomial-time* GI *algorithm for graphs of bounded degree.*

We very briefly sketch Luks' algorithm. The graph isomorphism problem for graphs of bounded degree reduces to the computation of the automorphism groups of rooted graphs of bounded degree. Let X be such a rooted graph. Consider X_i, the subgraph that consists of those edges and vertices with distance at most i from the root. We successively compute the automorphism of X_{i+1}, from the automorphism of X_i. Let $\mathrm{Aut}(X_i) =: A_i$ be the automorphism group of this subgraph. As the group A_{i+1} operates on the set X_i we obtain a group homomorphism $A_{i+1} \to A_i$. The kernel of this map is the pointwise stabilizer of X_i in A_{i+1}. Thus generators for A_{i+1} may be computed by lifting generators of the image of A_{i+1} in A_i and computing generators of the kernel. Generators for the kernel are directly computable, but for the computation of the image of A_{i+1} one has to resort to group theory. This image is the stabilizer of the edges in X_{i+1} not contained in X_i. The property that makes these groups accessible is the fact that for all i, the composition factors of A_i are subgroups of S_d, where d is the maximum degree of X.

We do not go into further detail here, as it will divert us too much. The methods can also be used to obtain an algorithm, with moderately exponential running time, that canonically labels a graph [7]. Gary Miller [101] has generalized Luks' method to a (natural) algorithm that solves GI in polynomial time for a graph class that concurrently contains the graphs of bounded degree and the graphs of bounded genus.

This ends our rough overview over existing graph isomorphism algorithms. Next we introduce a new randomized algorithm that uses statistical tests to solve the graph isomorphism problem.

2.6 The ScrewBox

In this section we describe the ScrewBox algorithm, a randomized algorithm for GI that performs particularly well on pairs of graphs which are "very similar" but non-isomorphic. Given any two graphs, the algorithm either supplies an isomorphism, or concludes, with a selectable error probability, that the input graphs are not isomorphic.

A standard approach to detect whether two graphs are non-isomorphic is via graph invariants. A graph invariant is any function on graphs invariant under isomorphisms. Basic examples of invariants are the degree sequence, i.e., the (multi-)set of node degrees, or the set of degree sums of all neighbors of all nodes. Any combination of invariants is also an invariant. A possibly more expressive invariant computes the maximum flow between all pairs of vertices in a graph. If an invariant yields different values on two graphs, the graphs cannot be isomorphic.

On highly structured graphs, like the incidence graphs of finite projective planes, however, such simple predicates will not suffice. We obtain a very expressive invariant by considering the multiset of colors that is obtained by the k-dimensional Weisfeiler-Lehman refinement. The strength of this invariant is indicated by the fact that it solves GI on various graph classes, as shown by Theorems 3 and 4. However, excessively strong invariants are computationally far too expensive. To remedy this we construct invariants that can be evaluated in a probabilistic fashion. An easy example of this is the invariant that counts the number of triangles in a given graph. Assume two graphs on n vertices contain a different amount of triangles. When determining the number of triangles in both graphs, we observe different counts and infer that the graphs are not isomorphic. If these counts differ strongly, we can save time at the expense of certainty: We randomly sample triples of vertices in both graphs, and eventually note that the relative frequency of the triple forming a triangle differs in the two graphs. We conclude (with a certain error probability) that the graphs are not isomorphic.

Sampling triangles is good for many pairs of graphs, but will not suffice for pairs of equally large graphs with equal number of triangles. For these we require other invariants. The idea behind the algorithm that follows is to dynamically construct invariants that can be evaluated through statistical tests. Figure 2.6 depicts from a high level view point how the stochastic algorithms, that we design, work.

The specific algorithm we now describe first appeared in [79] and was developed together with Martin Kutz.

Intuitively, the algorithm tries to find certain patterns in the input graphs by sequentially sampling nodes in a randomized fashion. The goal is to observe significantly

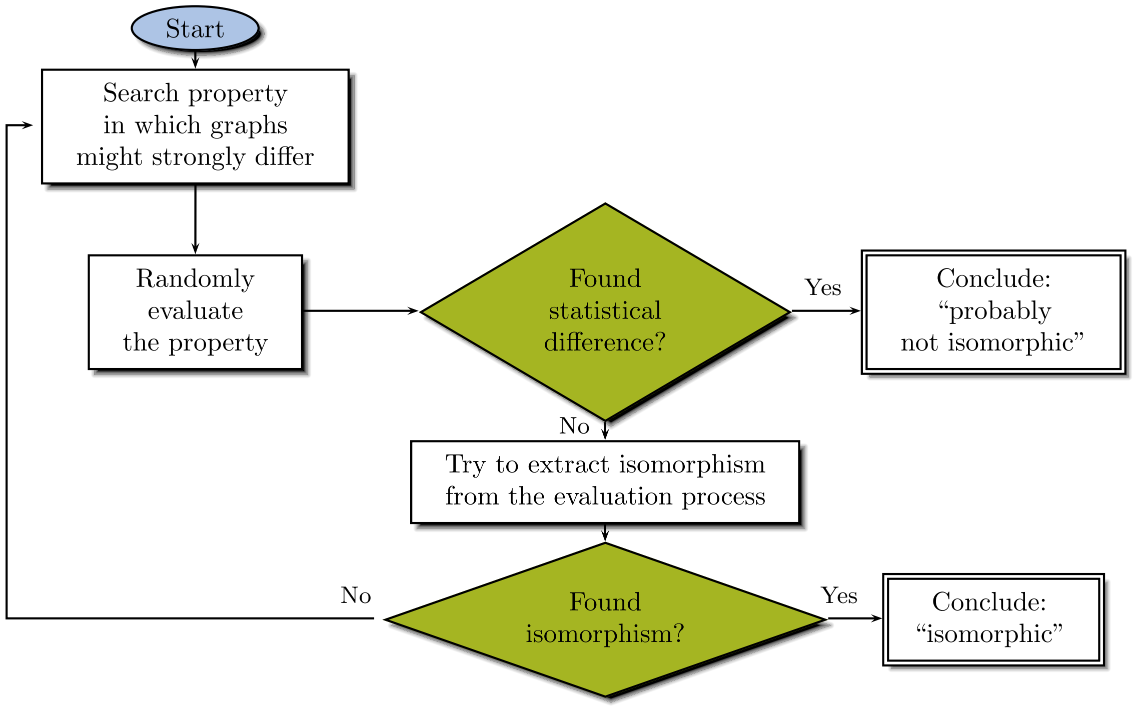
Start
Search property in which graphs might strongly differ
Randomly evaluate the property
Found statistical difference?
Yes
Conclude: “probably not isomorphic”
No
Try to extract isomorphism from the evaluation process
Found isomorphism?
No
Yes
Conclude: “isomorphic”

Figure 2.6: High level view of the stochastic GI algorithms, such as the ScrewBox algorithm

different behavior of this sampling process for the two given graphs. A single sample run draws nodes $s_1, s_2, \dots, s_n$, the *sample*, one after another, where each s_t has to fulfill a certain set of rules. Such a rule determining the admissibility of a sample node s_t is called a *screw*. By replacing the screws with other screws, the sampling process can be steered. Specifically, in each step t, a set of screws determines the set A_t of admissible nodes, from which s_t is drawn at random. Then the sampling proceeds to vertex s_{t+1}. If, for the first time, the set A_T is empty, for some $T \in \{1, \dots, n\}$, the sampling terminates and we record the length T at which this happened. If, after running this process many times, the frequencies of these termination lengths differ significantly for samples on two given graphs, we conclude that (with high probability) the graphs are not isomorphic.

The collection of all screws for all lengths $1, \dots, n$ is called the *screw box* (as opposed to the word “ScrewBox,” which denotes the complete algorithm). The construction of the screw box and the selection and tuning of the screws is a complex dynamic process that forms the core of the ScrewBox and will be subsequently described in this section.

Throughout this section, for any graph G, we denote by $\lambda_G \colon V^2 \to \{-1, 0, 1\}$ the *characteristic edge function*, that is $\lambda_G(v_1, v_2) = 1$ if v_1 and v_2 are adjacent, $\lambda_G(v_1, v_2) = -1$ if $v_1 = v_2$ and 0 otherwise. For the characteristic edge function, we

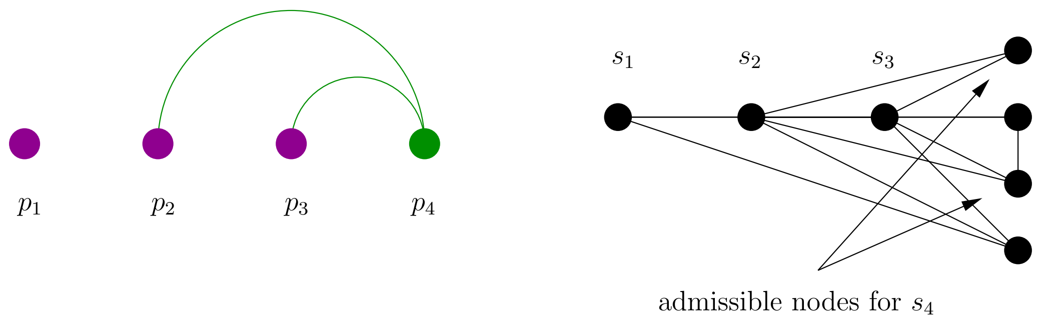

Figure 2.7: Depiction of the 0-level screw $S^{4,0}$ with result $(0,1,1)$, when evaluated on the pattern p_1, p_2, p_3, p_4 (left), and the corresponding admissible nodes for s_4 in a graph of size 7, in which s_1, s_2, s_3 is the previously chosen sample, (right).

liberally omit the parameter G, that specifies the graph, whenever it is evident from the context.

Definition 19 (screw). A *screw* applicable at length t is a function $S\colon \mathcal{G} \times V^t \to M$ invariant under graph isomorphism that assigns t-tuples of vertices of a graph some value in a set M.

Thus if S is a screw and $(v_1, \dots, v_t) = \bar{v}$ and $(v'_1, \dots, v'_t) = \bar{v}'$ are ordered tuples of vertices in two equally large graphs G and G' respectively, then $S(G, \bar{v}) = S(G', \bar{v}')$ if there is an isomorphism from G to G' that maps $\bar{v}$ to $\bar{v}'$, (as ordered tuples). For screws, we omit the parameter G whenever it is evident from the context (as we do for the characteristic edge function).

We now define the most basic of screws that will be used in the algorithm:

Definition 20 (0-level screw). For any colored graph and any tuple of vertices $(v_1, \dots, v_t) = \bar{v}$ define $S^{t,0}(G, \bar{v}) := \big(\lambda(v_1, v_t), \dots, \lambda(v_{t-1}, v_t)\big)$, the *0-level screw* of length t.

The 0-level screw thereby encodes the adjacency type of the vertex v_t with the vertices $v_1, \dots, v_{t-1}$, taking their order into account.

Fact 1. *A 0-level screw can be computed in linear time (more precisely in can be computed in $\mathcal{O}(\max\{t, n\})$), as it only involves $t-1$ edges incident with v_t.*

For illustrative purposes, we now develop a basic version of the algorithm that only uses these 0-level screws and a very simple statistical test.

2.6.1 The basic sampling algorithm

Given two input graphs G_1 and G_2, the basic sampling proceeds in the following way:

If the graphs are not of the same size n we declare G_1 and G_2 as not isomorphic due to their size. Otherwise, (which we implicitly assume from now on), we pick an arbitrary permutation $\bar{p} = p_1, p_2, \ldots, p_n$ of the vertices of the graph G_1. This permutation is called the *pattern* and will be fixed for the rest of the algorithm. Next we initialize a *histogram*, a map $H \colon \mathbb{N} \times \{1,2\} \to \mathbb{N}$, as the constant 0 map. By $H_j(i)$ with $j \in \{1,2\}$ we denote the value $H(i,j)$. Then alternating for both graphs we repeat the following: We pick a random vertex s_1. When s_{i-1} has been picked, we find a vertex s_i, by repeatedly drawing vertices uniformly at random (without replacement) from the vertices of G, until we find a vertex that is admissible, i.e. a vertex v that satisfies $S^{i,0}(p_1, \ldots, p_i) = S^{i,0}(s_1, \ldots, s_{i-1}, v)$.

Figure 2.7 illustrates the 0-level screw $S^{4,0}$. With its evaluation of

$$S^{4,0}(G_1, \bar{p}) = \big(\lambda(p_1, p_4), \lambda(p_2, p_4), \lambda(p_3, p_4)\big) = (0, 1, 1)$$

on the pattern $\bar{p}$, it filters, for a sample s_1, s_2, s_3 all vertices that are not adjacent to s_1 but adjacent to s_2 and s_3, as candidates for s_4, i.e., all vertices s for which $S^{4,0}(G_j, (s_1, s_2, s_3, s)) = S^{4,0}(G_1, \bar{p})$.

If an admissible vertex has been found we increase i and continue by drawing the next admissible vertex s_{i+1} for the sample. Otherwise we mark the length $T := i$ at which the sampling process could not be prolonged, by increasing $H_j(i)$ where j is 1 or 2 depending on whether the sampling was taken from G_1 or G_2 respectively. The sampling process is repeatedly performed alternately in the two input graphs G_1 and G_2: Thus a sample $s_1, \ldots, s_T$ is drawn from graph G_1, then a sample $s_1, \ldots, s_{T'}$ is drawn from G_2, and then again a sample $s_1, \ldots, s_{T''}$ is drawn again from graph G_1, and so on.

The sampling process induces two random variables h_1 and h_2, where $h_1 = i$ (respectively $h_2 = i$) is the observation that a single sampling in G_1 (respectively G_2) terminates with length i.

Theorem 8. *The sampling process constitutes for each graph G_j (with $j \in \{1,2\}$) a random variable h_j with values in $\mathbb{N}$, for which the outcome is the length of the sample that has been drawn. These random variables have equal distribution if and only if the graphs are isomorphic.*

Proof. Since the 0-level screws are invariant under graph isomorphism and, furthermore, in every step a vertex is chosen uniformly at random among the set of admissible vertices (those that have a certain value when the screw $S^{i,0}$ is evaluated), the whole sampling process is invariant under graph isomorphism.

If at some point a sample of length n is found for graph G_2, an isomorphism is found. It is given by the map that maps $p_i \mapsto s_i$. This can only (and will) happen with positive probability if G_2 is isomorphic to G_1. Contrarily, since the pattern has been taken from G_1, the random variable h_1 *always* attains the value n with positive probability. If the graphs are not isomorphic then h_1 and h_2 differ in the probability of the outcome n, i.e., they do not have the same distribution. □

If we continue taking samples from isomorphic graphs we therefore eventually, (i.e., asymptotically almost surely), encounter an isomorphism. When sampling in two non-isomorphic graphs, we have to content ourselves with performing a statistical test. If after a long period of time, i.e., when many samples have been drawn, we observe that the distributions of the random variables differ, we conclude, with some error probability, that the graphs are not isomorphic. One way of performing the statistical test is to only consider events for which h_1 or h_2 have n as their outcome. If this happens repeatedly for h_1 (which it must over time) but never for h_2, we conclude that the graphs are not isomorphic: Putting things together we obtain Algorithm 1 that uses exactly this event as stopping criterion.

Algorithm 1 The basic sampling algorithm

Input: Two graphs G_1, G_2 and the acceptable probability of error α

Output: **Yes**, if $G_1 \cong G_2$
No, if $G_1 \ncong G_2$, or with probability of at most α if $G_1 \cong G_2$

```
1: if $G_1$ and $G_2$ have different size then
2:   return No
3: end if
4: $n \leftarrow |G_1|$
5: initialize $H_j(i) \leftarrow 0$ for $j \in \{1,2\}$ and $i \in \{1,\dots,n\}$
6: pick a random permutation $(p_1,\dots,p_n)$ of $V(G_1)$                // the pattern
7: repeat
8:   for $j \in \{1,2\}$ do                          // perform sampling in each graph
9:     $i \leftarrow 1$
10:    while $i \leq n$ and there is $s$ with $S^{i,0}(G_1,p_1,\dots,p_i) = S^{i,0}(G_j,s_1,\dots,s_{i-1},s)$
       do
11:      pick a random $s$ that satisfies this equation
12:      $s_i \leftarrow s$
13:      $i \leftarrow i+1$
14:    end while
15:    $H_j(i) \leftarrow H_j(i)+1$
16:  end for
17: until $H_2(n) > 0$ or $2^{-H_1(n)} < \alpha$
18: if $H_2(n) > 0$ then
19:   return Yes
20: else
21:   return No
22: end if
```

The running time of the algorithm essentially depends on the number of sampling processes that have to be performed. Each such sampling process takes $\mathcal{O}(n^3)$ time: As i ranges from 1 to a maximum of n, at most $2n^2$ screws are evaluated. Each evaluation can be done in linear time, as observed in Fact 1. (As the outcome of the screws evaluated on the pattern remains the same throughout the duration of the algorithm, we precompute these values. Per sampling we then evaluate at most n^2 screws.)

Lemma 1. *The expected number of samplings that are performed by Algorithm 1 is in* $\mathcal{O}(\frac{n!}{|\operatorname{Aut}(G_1)|} \cdot \log_2(1/\alpha))$.

Proof. The probability that for $j = 1$ the sampling process continues up to $i = n$ is related to the number of permutations that yield an isomorphic graph. There are $|\operatorname{Aut}(G_1)|$ such permutations. If the first i vertices of such a permutation have been sampled, the next vertex of the permutation will be drawn with a probability of at least $\frac{n-i}{n}$. We note that vertices cannot occur twice in a sample, as 0-level screws prohibit this. Therefore every specific permutation will occur with probability of at least $\frac{1}{n!}$ and these events are disjoint. The total probability is therefore at least $\frac{|\operatorname{Aut}(G_1)|}{n!}$. If this event happens $\lceil \log_2(1/\alpha) \rceil$ times then termination condition of the algorithm applies. The number of samplings needed for the event to happen once is geometrically distributed and in expectation $\frac{n!}{|\operatorname{Aut}(G_1)|}$. By linearity of the expected value we conclude the result. (On isomorphic instances the process will end after an expected $\mathcal{O}(\frac{n!}{|\operatorname{Aut}(G_1)|})$ samplings due to samplings in G_2.) □

This upper bound on the number of samplings will hold for any version of the algorithm that we present in this thesis. Together with the running time for performing one sampling process we conclude:

Theorem 9. *Given input graphs G_1 and G_2, the expected running time of Algorithm 1 is in* $\mathcal{O}(n^3 \cdot \frac{n!}{|\operatorname{Aut}(G_1)|} \cdot \log_2(1/\alpha))$.

Proof. The proof is immediate from Lemma 1 and the aforementioned fact that the time required for each sampling is in $\mathcal{O}(n^3)$. □

We will now convince ourselves, that the algorithm satisfies the specified error probability. If the outcome is **Yes** then indeed an isomorphism has been found and no error has occurred. If the output is **No**, then the algorithm only erred if the graphs are isomorphic. If the graphs are isomorphic then the events $h_1 = n$ and $h_2 = n$ are equally likely. Thus the probability that $h_1 = n$ is observed k times before the event $h_2 = n$ is observed even once is bounded by 2^{-k}. (Note that since we always perform two samplings, for the test the samplings may be considered as being performed simultaneously in both graphs). If the output is **No** then $H_2(n) = 0$ but $2^{-H_1(n)} < \alpha$. Thus, by choosing $k = H_1(n)$, we conclude the probability of error, i.e., the significance level of the test, is bounded by α.

The algorithm in this simple form only exploits the values of $H_j(n)$. Naturally it is possible to include other values of the histogram into the algorithm: To see this, we use the introductory example of sampling triangles. We assume for the moment that G_1 and G_2 are d-regular graphs, and that in the one graph G_1 most pairs, say c_1, of adjacent vertices have a common neighbor while in G_2 only $c_2 < c_1$ vertices have a common neighbor. We assume further that the pattern vertices p_1, p_2 and p_3 form a triangle. Since the graphs are regular, the sampling process picks the first two vertices s_1 and s_2 uniformly at random among the pairs of vertices that form an edge in the graph. The probability that the sample can be prolonged to s_3 depends on s_1

and s_2 having a common neighbor. More precisely the probability that a sample in G_1 can be prolonged is $\frac{2 \cdot c_1}{d \cdot n}$ as opposed to $\frac{2 \cdot c_2}{d \cdot n}$ in graph G_2, so the probability for h_1 and h_2 to yield a value of 3 or more differs by at least $\frac{1}{d \cdot n}$, which is statistically (a lot) more significant than the bound we used for the difference in probability of the events $h_i = n$. We defer further treatment of how to test for significant difference of the random variables h_1 and h_2 to Section 2.7.

This example also shows that we can get better upper bounds for running times when we consider restricted graph classes (easy graphs). Such considerations will be postponed until we have an enhanced version of the algorithm.

The basic sampling algorithm may be altered by removing the check whether G_1 and G_2 have the same size, so that it can be used for subgraph isomorphism detection. This problem is known to be $\mathcal{NP}$-complete. We therefore do not expect to find a satisfying bound for the algorithm in general. The modifications we perform on the basic sampling algorithm, however, prevent it from being used for this purpose. (This is relieving as the seemingly easier problem GI should not necessarily be solved with a reduction to an $\mathcal{NP}$-complete problem.)

2.6.2 Higher level screws

With highly structured graphs, this basic sampling algorithm leads to unacceptably long running times. Our way of improving the performance is to introduce vertex invariants into the algorithm, that more strongly capture the structure of a graph. As mentioned earlier, a screw in the screw box can, in principle, be an arbitrary predicate invariant under graph isomorphism that determines whether a vertex v_t is a valid extension of the sample $v_1, \dots, v_{t-1}$ in G.

The screws of higher level, that we now define, do not only consider the incidence structure of the sample itself but take into account its relative position within the rest of the graph. As a result, the average sample length may increase. This should, however, be seen only as a side effect since our main goal is to increase the statistical significance. The performance depends on the "degree of non-isomorphism" between the given graphs. Strong similarity makes non-isomorphism verification difficult, which then requires more computation time.

Thus, in order to improve the performance, we recursively design a set of vertex invariants, the k-level screws. Intuitively, for some level k these invariants compute in advance all possible extensions of the sample by k further vertices, and keep track of how the adjacency structure to the sample and amongst the chosen vertices is composed.

The definition extends Definition 20, which defines the 0-level screws.

Definition 21 (k-level screw). For any colored graph G, we define the *k-level screw* $S^{t,k}$ by their evaluation on any t-tuple of vertices $\bar{v} = (v_1, \dots, v_t)$ recursively as:

$$S^{t,0}(G, \bar{v}) := \big(\lambda(v_1, v_t), \dots, \lambda(v_{t-1}, v_t)\big),$$

$$S^{t,k}(G, \bar{v}) := \big\{\big\{S^{t,0}\big(G, (v_1, \dots, v_t)\big)\big\}\big\} \cup \big\{\big\{S^{t+1,k-1}\big(G, (v_1, \dots, v_t, u)\big) \mid u \in G\big\}\big\}.$$

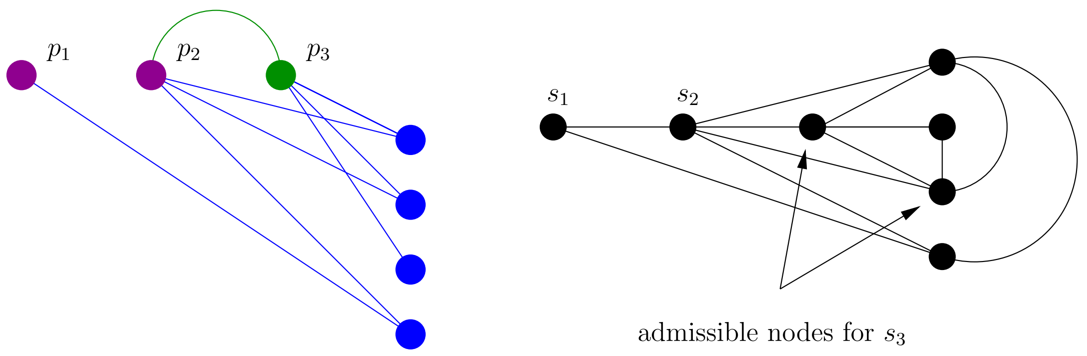

Figure 2.8: The figure depicts the 1-level screw $S^{3,1}$ which has an evaluation of $\{\{(0,1)\}\} \cup \{\{(0,1,1),(0,1,1),(0,0,1)(1,1,0),(-1,0,0),(0,-1,1),(0,1,-1)\}\}$ on the pattern p_1,p_2,p_3 (left), and the corresponding admissible nodes for s_3 in a graph of size 7, in which s_1,s_2 is the previously chosen sample (right).

By this definition $S^{0,0}$ is the constant function that evaluated to the empty tuple (). While the values of 0-level screws are tuples, the values of k-level screws are multisets of values of screws of lower level. Again, as for 0-level screws, we frequently drop the first parameter, the graph G, if it is evident from the context.

Figure 2.8 depicts an evaluation of a 1-level screw. It shows the 1-level screw $S^{3,1}$ applied to a sample s_1,s_2 in order to determine an admissible vertex s_3, which has to yield the value

$$S^{3,1}(G,(s_1,s_2,s_3)) = \{\{S^{3,0}(G,(p_1,p_2,p_3))\}\} \cup \{\{S^{4,0}(G,(p_1,p_2,p_3,u)) \mid u \in G\}\} =$$
$$\{\{(0,1)\}\} \cup \{\{(0,1,1),(0,1,1),(0,0,1)(1,1,0),(-1,0,0),(0,-1,1),(0,1,-1)\}\}.$$

On the left the figure displays (in green) the adjacency pattern that the next vertex must exhibit. Also on the left (in blue) it displays the set of adjacencies, which vertices on the second level must form, after a third vertex has been chosen for the sample. (The nodes that are already in the pattern, which correspond to tuples that contain -1, have been depicted in the figure.)

For screws in general we observe, if $S^{t,k}(G,\bar{v}) = S^{t,k}(G',\bar{v}')$, then $S^{t,k-1}(G,\bar{v}) = S^{t,k-1}(G',\bar{v}')$. The screws thus form a set of invariants that increase in strength with increasing level k. Two graphs of the same size G and G' are isomorphic if and only if $S^{0,n}(G) = S^{0,n}(G')$. The screws can be used to characterize various regularity conditions on graphs: A graph is regular if and only if $S^{1,1}(G,\cdot)$ is constant. If additionally the value of $S^{2,1}(G,v_1,v_2)$ only depends on the value of $S^{2,0}(G,v_1,v_2)$ then the graph is strongly regular (see Subsection 2.8.1). A graph is vertex-transitive if and only if $S^{1,n-1}(G,\cdot)$ is constant.

Since the map $S^{t,k}(G,\cdot)$ is an isomorphism invariant on the t-tuples of vertices, they may be used to construct a refinement procedure: refine a coloring of the t-tuples by partitioning them according to their value on $S^{t,k}(G,\cdot)$ and repeat this step until the partition corresponding to the coloring stabilizes.

We now briefly mention, without proof, two interpretations of the screws in other mathematical contexts. Namely, they can be understood with combinatorial games, as well as in the context of logic.

It is known that for the k-dimensional Weisfeiler-Lehman refinement there exists a corresponding Ehrenfeucht-Frässé game [23]. We will not go into detail of the theory. Consult Spencer's book on random graphs [120] for an introduction to these games in the context of graph theory. We do mention, however, that there is a corresponding Ehrenfeucht-Frässé game in our case. The game has to be adapted in such a way that the spoiler may reuse the pebbles, but may only do so in a last-in-first-out order, i.e., the player first has to remove pebbles that were put onto the graph last.

As for the invariants that are used by the Weisfeiler-Lehman vertex refinement, there is also a connection between the k-level screw refinement and statements expressible in formal logic. From this we deduce a relationship between the refinement with screws and the Weisfeiler-Lehman refinement:

The k'-dimensional Weisfeiler-Lehman vertex refinement, as given in Definition 14, is at least as fine as the refinement with k-level screws on t-tuples, if $k + t \leq k' + 1$. This can be seen by expressing $S^{t,k}$ as a sentence in first order logic. In particular, the screw $S^{t,k}$ is a first order sentence in the language with ordered quantifiers, i.e., the set of clauses that use constants $v_1, \dots, v_t$ that represent vertices in the graph and quantified vertex variables $x_1, \dots, x_n$ (with counting), where nesting of the variables occurs in a fixed order in any clause of the formula. In fact, the screw $S^{t,k}$ can distinguish any two t-tuples of vertices that can be distinguished by a sentence in this logic. For the logical statements corresponding to the Weisfeiler-Lehman refinement, there is no restriction on the ordering in which the variables occur.

Consequently, for any fixed level k, the graphs obtained with the CFI-construction cannot be distinguished with the vertex invariants of level k. This answers a question Martin Kutz and I posed in [79] in the negative. In fact, the Weisfeiler-Lehman refinement is strictly finer than the refinement with k-level screws. As an easy example, consider two graphs each consisting of two disjoint cycles $C_n \cup C_m$ and $C_{n'} \cup C_{m'}$ such that $n+m = n'+m'$. For n, m, m', m' large enough, $S^{1,k}$ will not be able to distinguish vertices in the graphs, while the 2-dimensional Weisfeiler-Lehman refinement partitions the vertices of these graphs into the orbits. However, the screws offer significant advantages over the Weisfeiler-Lehman refinement. Before we explain these, we first need to know how fast and with how much space requirement a screw can be evaluated. As a value of a screw of level k consists of n values of screws of level $k-1$, evaluating the screws the way they are defined yields intractable values. In practice, we therefore hash all screw values to integers, a multiset of n integers thus hashes again to an integer. We use a hash function, which enables us to compute the hashed value for a multiset of n integers in linear time in n. With this we can space-efficiently compute the screws:

Theorem 10. *Given vertices $v_1, \dots, v_t$, a (hashed) evaluation of $S^{t,k}(v_1, \dots, v_t)$ can be performed in $\mathcal{O}\left((\max\{t+k, n\})^k\right)$ time. This computation requires $\mathcal{O}(n)$ space.*

Proof. This can be seen by induction over k, starting with Fact 1 that $S^{t,0}(v_1, \dots, v_t)$

can be computed in linear time. To compute $S^{t,k}(v_1, \ldots, v_t)$ one has to perform n computations $S^{t+1,k-1}(v_1, \ldots, v_t, u)$, of screws of level $k-1$, and one computation of $S^{t,0}(v_1, \ldots, v_t)$. Together this yields a running time of $\mathcal{O}\big(n \cdot (\max\{t+k, n\})^{k-1} + \max\{t, n\}\big) \subseteq \mathcal{O}\left((\max\{t+k, n\})^k\right)$. With the straight forward depth-first evaluation, we attain the desired space requirement. $\square$

Since we only deal with the case where t and k add to a value of at most n, the computation time simplifies to $\mathcal{O}(n^k)$. As previously mentioned, the k-level screws take the whole graph into account. Therefore, with the enhanced algorithm we do not implicitly solve the subgraph isomorphism problem. (See the remark at the end of Subsection 2.6.1). We emphasize that the main goal when designing the screw box is *not* to find very long samples but to create a significant deviation in the termination levels on the two graphs, and thus to find proof for non-isomorphism.

The sampling algorithm may use any graph invariant to determine whether a vertex is admissible to extend the sample. By using the k-dimensional Weisfeiler-Lehman refinement to decide admissibility, we have developed a graph isomorphism algorithm that distinguished all non-isomorphic graphs, as opposed to the Weisfeiler-Lehman algorithm described in Section 2.3, which fails on certain non-isomorphic graphs. Since the computation time of the k-level screws matches that of one step in the Weisfeiler-Lehman coloring procedure, but the refinement corresponding to the latter is provably finer, the questions arises: Why are the screws introduced?

The reason for this is threefold. First, a k-level screw can be used to evaluate whether a vertex is admissible, independent of the admissibility of other vertices. Since cheaper screws may already exclude many candidates, we only have to evaluate expensive screws on a few candidates. If, on the other hand, we have found an admissible vertex, no other vertex has to be tested. Second, for all we know the Weisfeiler-Lehman refinement requires $\Omega(n^k)$ space, where a k-level screw can be computed in $\mathcal{O}(n)$ space. This space requirement renders the higher dimensional Weisfeiler-Lehman refinement infeasible. Third, there is a way to compute only partial, yet conclusive, information for a screw. This can be done in an organized fashion, which we elucidate in the following subsection.

2.6.3 Cheap screws of high level

Naïvely evaluating k-level screws repeatedly the way they are defined soon becomes impractical, even for 2-level screws. We therefore optimize the screws by stripping them of the consideration of irrelevant nodes (many nodes turn out to have no effect on the value of a screw) and of superfluous adjacency tests. Eventually, we are able to work with highly fine-tuned screws that have very good separation properties at low computational cost. This screw tuning is an integral part of the algorithm and is indispensable for achieving acceptable running times. This is also reflected in the code, since the part that computes the cheap screws has been optimized the most.

For most sample lengths there is no need to employ a screw of higher level at all. For those lengths at which higher level screw are required, the information computed by the screw exceeds what is needed to differentiate vertices. If $S^{t,k}(G_i, \bar{s}) \neq S^{t,k}(G_1, \bar{p})$ then one of the following two cases must occur: Either $S^{t,0}(G_i, \bar{s}) \neq S^{t,0}(G_1, \bar{p})$, in which case we do not need to resort to a high level screw in the first place, (we could use the screw $S^{t,0}$), or the multisets computed by the screws differ in the frequency of some element $U = S^{t+1,k-1}(G_1, (\bar{p}, u))$. In this case we define a new screw $R^{t,k}(G, \bar{v}) = \{\!\{S^{t+1,k-1}\big(G, (\bar{v}, u)\big) \mid u \in V(G),\, S^{t+1,k-1}(G, (\bar{v}, u)) = U\}\!\}$. Under the previous assumption we obtain $R^{t,k}(G_i, \bar{s}) \neq R^{t,k}(G_1, \bar{p})$, and conclude that the screw $R^{t,k}$ also differentiates $\bar{s}$ and $\bar{p}$. To compute $R^{t,k}(G, \bar{v})$ we may dispose of all extension vertices u for which we can show that $S^{t+1,k-1}(G, (\bar{v}, u)) \neq U$. When computing the value of the refined screw $R^{t,k}$ we first check whether a vertex u has the correct adjacencies with $\bar{v}$, before computing the recursive structure of $R^{t,k}$.

Now that we have excluded vertices on the first recursive level, we further reduce the computation time by excluding vertices on the second level that do not exhibit specific adjacencies with $\bar{v}$. This is not always possible. If possible we then continue the exclusion of vertices for levels beyond the second.

Definition 22 (cheap screw). The set of *cheap screws* is recursively defined by:

- For all $t, k \in \mathbb{N}$ the screw $S^{t,k}$ is a cheap screw of level k and length t.
- For any set of cheap screws $R_1^{t+1,k-1}, \ldots, R_\ell^{t+1,k-1}$ of level $k-1$ and length $t+1$, and any set $U = \{U_1, \ldots, U_\ell\}$ of possible values of these screws, the screw $R^{t,k}$ given by $R^{t,k}(G, \bar{v}) :=$

$$\bigcup_{i \in \{1,\ldots,\ell\}} \{\!\{R_i^{t+1,k-1}\big(G, (v_1, \ldots, v_t, u)\big) \mid u \in V(G),\, R_i^{t+1,k-1}(G, (\bar{v}, u)) = U_i\}\!\}.$$

  is a cheap screw.

We give an intuition of the information computed by the cheap screws: The k-level screws $S^{t,k}$ are the invariants that broadly determine how individualization of k further vertices will affect the graph. More precisely, they capture the recursive individualization tree and all information in it (including all adjacencies and the whole recursive structure). The cheaper screws do the same, but only on a smaller portion of the tree, i.e., they only continue the recursion on vertices that meet certain adjacency requirements. An efficient implementation of the cheap screws requires us to precompute lists of the vertices that meet the adjacency requirements. For example, when choosing a vertex for the first level, we update a list of vertices that have the correct adjacency requirement for every level beyond the first. These lists for each level will be updated as the recursion level changes. For further details we refer the inquisitive reader to the code [116], and conclude with an example:

We define the two circulant graphs G_1 and G_2 on the vertices $V := \{1, \ldots, 15\}$. For two vertices $v_1, v_2 \in V$, we define the distance modulo 15 as the number given by $|v_1, v_2|_{15} := \min\{|v_1 - v_2|, 15 - |v_1 - v_2|\}$. The edge sets of the graphs are given by $E_1 := \big\{\{v_1, v_2\} \mid v_1, v_2 \in V \wedge |v_1, v_2|_{15} \in \{1, 5\}\big\}$ for G_1 and respectively by

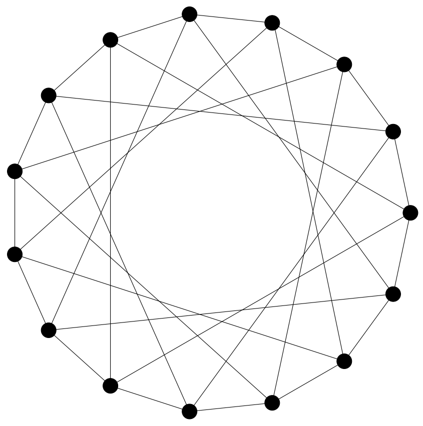
Figure 2.9: The circulant graph on 15 vertices with neighbors at distances 1 and 5

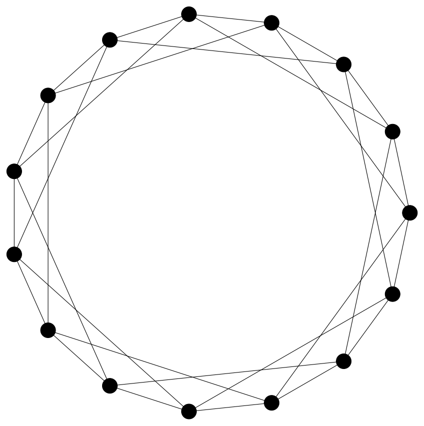
Figure 2.10: The circulant graph on 15 vertices with neighbors at distances 1 and 3

$E_2 := \big\{\{v_1, v_2\} \mid v_1, v_2 \in V \wedge |v_1, v_2|_{15} \in \{1, 3\}\big\}$ for G_2. These graphs are depicted in Figures 2.9 and 2.10.

As circulant graphs, they are vertex transitive. In particular they are 4-regular and the map $S^{1,k}$ is constant for any k. Moreover, given two adjacent vertices v_1, v_2 in one of the graphs, there are, besides v_1 and v_2, exactly three vertices adjacent only to v_1, three vertices adjacent only to v_2, and seven $(15 - 2 \cdot 3 - 2 = 7)$ vertices adjacent to neither v_1 nor v_2. This means, in particular, that there is no way to differentiate pairs of adjacent vertices by only considering one additional vertex. In other words, the 1-level screws $S^{2,1}(G_1, \cdot\,)$ and $S^{2,1}(G_2, \cdot\,)$ are constant and equal on pairs of adjacent vertices. Thus, when trying to differentiate the graphs with the 2-level screw $S^{1,2}$, we choose $R^{1,2}$ given by $R^{1,2}(v_1) = \{\{S^{2,1}(G_1, v_1, v_2) \mid v_1 \text{ not adjacent to } v_2\}\}$. When we evaluate $R^{1,2}$ on graph G_2, we see that for every vertex v_1 in G_2 there is a non-adjacent vertex v_2, such that v_1 and v_2 have three common neighbors. (This corresponds to finding two cycles of length 4). When we evaluate $R^{1,2}$ on the graph G_1, however, we see that this is not the case for any vertex in G_1. We thus further restrict the screw on the second level to only consider vertices v_3 that are adjacent to v_1 and v_2. To conclude that the graphs are not isomorphic, it thus suffices to count, in both graphs, how many non-adjacent vertices have exactly three common neighbors.

2.6.4 Customizing the algorithm

We have seen that the basic sampling algorithm is very customizable and there are lots of design choices to be made. In particular we have a variety of choices for the invariants (screws) we use, for the refinement techniques we want to apply and for the choice of the pattern (which corresponds to an individualization strategy).

Algorithm 2 The generic (enhanced) sampling algorithm (with options marked in red)

Input: Two graphs G_1, G_2 and a significance level α

Output: **Yes**, if $G_1 \cong G_2$
No, if $G_1 \ncong G_2$, or with probability of at most α if $G_1 \cong G_2$

```
 1: if $G_1$ and $G_2$ have different size then
 2:     return No
 3: end if
 4: initialize $H_j(i) = 0$ for $j \in \{1,2\}$ and $i \in \{1,\dots,n\}$
 5: pick a random permutation $(p_1,\dots,p_n)$ of $V(G_1)$        // the pattern
 6: repeat
 7:     for $j \in \{1,2\}$ do
 8:         sample each graph according to some rules (vertex invariants, vertex refine-
            ments, rule for individualization)
 9:         update the histogram $H_i$ accordingly
10:     end for
11: until some chosen statistical test on $H_1$ and $H_2$ will provide an answer with
    confidence $\alpha$
12: if the test is met then
13:     return Yes
14: else
15:     return No
16: end if
```

Section 2.7 explains statistical tests well-suited for the algorithm. Algorithm 2 shows (highlighted in red) where changes can easily be implemented into the basic sampling algorithm. These elements, which may be edited, are *options* with which we may run the algorithm.

Basically the algorithm may be run with any set of options also applicable in a refinement individualization algorithm such as Nauty. The statistical test has to be chosen in a way that guarantees the significance level desired by the user. The challenge is, on a given input, to dynamically find a set of options such that:

- significant statistical data is obtained, so that the test ends early,
- the sampling procedure remains computationally cheap and
- simplicity is maintained (theoretically and practically).

There is a trade-off between these three goals: When we increase statistical significance by invariants that are more powerful, each sampling process requires more time, as it has to evaluate these invariants. The more sophisticated our techniques, the more complicated they are. We thus have to find a balance between the goals to achieve our desired performance.

Our method for this is of adapting the options to a given pair of input graphs. We first focus on the possibility to adapt the admissibility rules to input graphs, and then elaborate on the choice of pattern, which corresponds to the individualization strategy. We deal with the options for the statistical test in Section 2.7. The admissibility rules are governed by the choice of where and what kinds of screws we use in the algorithm.

2.6.5 Placement of the screws

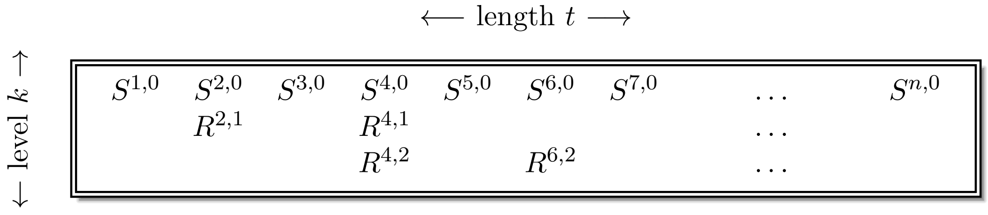
$\longleftarrow$ length t $\longrightarrow$
$\longleftarrow$ level k $\longrightarrow$
$S^{1,0}$ $S^{2,0}$ $S^{3,0}$ $S^{4,0}$ $S^{5,0}$ $S^{6,0}$ $S^{7,0}$ ... $S^{n,0}$
$R^{2,1}$ $R^{4,1}$ ...
$R^{4,2}$ $R^{6,2}$ ...

Figure 2.11: A typical screw box for graphs of size n: the screws $S^{t,0}$ are present at every length t, and at some lengths additional cheap screws $R^{t,k}$ of level k have been inserted.

In the previous subsection we have seen that the sampling algorithm is highly customizable. Even if we restrict ourselves to using screws of level at most 2, there are still lots of choices open. (Screws of arbitrary level will be too expensive, and therefore, in practice, we have to restrict the used level.) In the following we deal with the decision how and where (i.e., at what length t) to put what kind of screw. This is something that shall be done by the algorithm and not by the user.

Part of the construction process of the screw box is the continued evaluation of its quality. Insertions, deletions, and modifications of screws are meant to increase the statistical significance of the sampling and reduce the running time of the algorithm.

Usually, we have to deal with a trade-off between more expensive screws and sampling significance. This means that for an efficient sampling, the selection and placement of the screws has to be done with great care. A main feature of the algorithm is the self-adaptive behavior of the screw box's construction process. On a "simple" instance, for example, no expensive screw gets installed, whereas a highly structured graph induces a few expensive screws at crucial sample lengths. This way there is no need to specify in advance the difficulty or special properties of the input graphs. During the construction process, sampling sequences are taken from the input graphs, and note is taken of which screws are effective:

Definition 23 (effective, ultimate). We say that a screw $S^{t,k}$ is *effective* on the sample $\bar{v} = (v_1, \dots, v_{t-1})$ from the graph G with respect to some graph G' with pattern $\bar{p} = (p_1, \dots, p_t)$ of vertices from $V(G')$, if there is a vertex v_t such that $S^{t,k}(G, (\bar{v}, v_t)) \neq S^{t,k}(G', \bar{p})$.

We say that a screw $S^{t,k}$ is *ultimate* on the sample $\bar{v} = (v_1, \dots, v_{t-1})$ from the graph G with respect to some graph G' with pattern $\bar{p} = (p_1, \dots, p_t)$ of vertices from

$V(G')$, if for every vertex v_t it holds that $S^{t,k}(G,(\bar{v},v_t)) \neq S^{t,k}(G',\bar{p})$.

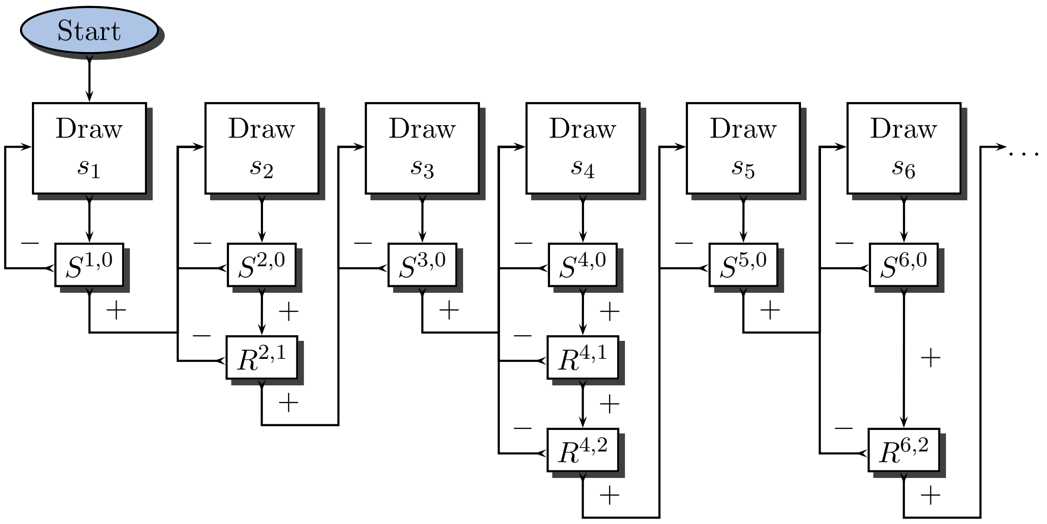

Figure 2.12: A flow diagram for a sampling performed with the screw box from Figure 2.11. For each screw there are two possible ways to continue, depending on whether the screw determines that the drawn sampling vertex is admissible (+) or not admissible (−). (It also depends on a previously chosen pattern $\bar{p}$, not depicted). If for some length t all candidates s_t have been rejected, the sampling terminates with length $T := t$, this value is supplied to the ScrewBox algorithm. (The algorithm then may choose to perform further samplings commencing at "Start").

Being effective thus says that the screw can determine for some candidates v_t, that they are not admissible. Being ultimate says that screw shows that no vertex is admissible.

We now describe the modifications on the set of screws, the screw box, that are applied in each sampling. The algorithm starts with a screw box that only contains one 0-level screw for every length of the sample. It thus starts with the set $\{S^{1,0}, S^{2,0}, \ldots, S^{n,0}\}$.

These screws will never be removed from the box, only screws of higher level will be inserted and possibly removed again. Since it is desireable for a given sample length to first evaluate cheaper invariants, followed by the more expensive ones, these 0-level screws will always be evaluated first. Figures 2.11 and 2.12 show an example of how the final screw box may be composed, and in what order they are evaluated during a sampling. The algorithm estimates the effectiveness of the screws, by counting the number of vertices that have been rejected by them. Screws are inserted and deleted according to the following rules:

Insertion and deletion rules for screws

Rule 1 If a screw rejects a vertex, the screw is asked whether there is a cheaper version, a child, of itself, that also rejects the vertex. If so, the cheaper version is placed at the same length as its parent. During a sampling, it is evaluated right before its parent.

Rule 2 If a screw is placed into the screw box, the effectiveness of any screw that during a sampling is used after the inserted screw is reset.

Rule 3 If the algorithm determines that a screw is not effective, it is removed from the box.

Rule 4 If a screw is ultimate, a screw of higher level is placed at a shorter length into the box. (If no screw of higher level is available, a screw of equal level is used). This process is done at most some constant number of times for any screw.

We need two observations that clarify how the effectiveness of different screws is related. The screws $S^{t,k}$ form an ordered set of invariants:

Theorem 11 (relational effectiveness of screws). *Let G' be a fixed graph, in which a pattern $\bar{p} = p_1, \dots, p_n$ has been chosen. Effectiveness of screws, for vertices $v_1, \dots, v_n$ in $V(G)$, has the following relational properties:*

1. *If $S^{t,k}$ is effective on $(v_1, \dots, v_{t-1})$, with respect to G' and $\bar{p}$, then for any $t' \geq t$ the screw $S^{t',k}$ is effective on any extension of the sample, with respect to G' and $\bar{p}$, i.e., it is effective on any tuple of the form $(v_1, \dots, v_{t-1}, \dots, v_{t'-1})$.*
2. *If for $t > 1$ a k-level screw $S^{t,k}$ is ultimate on all samples $(v_1, \dots, v_{t-2}, v)$ for any v in $V(G)$, with respect to G' and $\bar{p}$, then $S^{t-1,k+1}$ is ultimate on $(v_1, \dots, v_{t-2})$, with respect to G' and $\bar{p}$.*

Proof. To prove Statement 1 we assume there is a vertex $v \in V(G)$, such that

$$S^{t,k}(G, v_1, \dots, v_{t-1}, v) \neq S^{t,k}(G', \bar{p}).$$

We claim, that $S^{t',k}(G, (v_1, \dots, v_{t'-1}, v) \neq S^{t',k}(G', p_1, \dots, p_{t'})$. That is, we show that exactly this same vertex is not admissible for the longer sample. We consider two cases:

Case 1: If $S^{t,0}(G, v_1, \dots, v_{t-1}, v) \neq S^{t,0}(G', p_1, \dots, p_t)$, then

$$S^{t',0}(G, v_1, \dots, v_{t'-1}, v) \neq S^{t',0}(G', p_1, \dots, p_{t'}),$$

from which we conclude the claim.

Case 2: If the multiset $\{\{S^{t+1,k-1}(G, v_1, \dots, v_{t-1}, v, u) \mid u \in V(G)\}\}$ is different from the multiset $\{\{S^{t+1,k-1}(G', p_1, \dots, p_t, u') \mid u' \in V(G')\}\}$, then we conclude by induction on k, (the base case being Case 1), that the same holds for the multisets $\{\{S^{t'+1,k-1}(G, v_1, \dots, v_{t'-1}, v, u) \mid u \in V(G)\}\}$ and $\{\{S^{t'+1,k-1}(G', p_1, \dots, p_{t'}, u') \mid u' \in V(G')\}\}$, they thus also differ, which gives us the desired conclusion.

To prove Statement 2 we assume the contrary. In particular, we assume that there is a vertex $v' \in V(G)$ such that

$$S^{t-1,k+1}(G, v_1, \dots, v_{t-2}, v') = S^{t-1,k+1}(G', p_1, \dots, p_{t-1}).$$

But in this case, by definition, the multiset $\{\{S^{t,k}(G, v_1, \dots, v_{t-1}, u) \mid u \in V(G)\}\}$ and the multiset $\{\{S^{t,k}(G', p_1, \dots, p_{t-1}, u') \mid u' \in V(G')\}\}$ are equal. Thus for the vertex p_t there is a corresponding vertex v, such that

$$S^{t,k}(G, v_1, \dots, v_{t-1}, v) = S^{t,k}(G', p_1, \dots, p_{t-1}, p_t),$$

which yields a contradiction. □

The first part of the previous theorem tells us that for a given sample there is an earliest length t, at which a screw $S^{t,k}$ is effective. From that length onward the screws $S^{t',k}$ with $t' \geq t$ all are effective. This enables us to find the earliest length t for which the screw $S^{t,k}$ is effective via a binary search. The algorithm uses this technique whenever it applies Rule 4. The second part of the theorem tells us that if a sample cannot be prolonged anymore, employing a screw of higher level could have detected this at a shorter length.

These two observations motivate the rules for the placement of screws given above. The algorithm performs these rules until statistically significant data is obtained. When it is satisfied with the statistic outcome of the current screw box (see Subsection 2.7.4), the screw box is fixed and a statistical test is performed. (In Section 2.7 we develop methods to perform the statistical test, while the screw box is still being modified.)

2.6.6 Capabilities provided by the screws

The concept of rules that guide the placement process suggests an encapsulation of the screws themselves from the decision of their placement. In order for an outer algorithm to perform the choices regarding placement, the screws themselves have to supply certain information. We thus need an interface for the screws that enables access to the required information. This interface should in particular enable the screws to:

- supply a screw of equal level that may be inserted by the algorithm elsewhere in the box, i.e., at a different length.
- calibrate themselves: Given the pattern they should initialize themselves with values to which any sample is compared to, without having to resort to the pattern again.

- suggest a cheaper version of themselves, with which the algorithm may chose to replace the original screw, in order to save time.
- estimate their own effectiveness by determining how many sample vertices are rejected.
- evaluate their cost. One way of measuring this is to keep track of the number of accesses to the pairlabel matrix associated with the graph. See Subsection 2.9.2.
- suggest a stronger version of themselves in case the algorithm decides the screws used are too weak to generate significant data.

Assuming these capabilities, we may apply the rules from the previous subsection to effectively use few expensive screws at places they are needed.

The next option we investigate is the choice of the pattern, which was previously assumed to be arbitrary.

2.6.7 The choice of pattern

Depending on the input, the choice of the pattern may be of crucial importance. We give an example of an input where this is the case: We consider G_1^f and G_2^f, a pair of non-isomorphic graphs on which the algorithm is fast, i.e., a pair of graphs on which our algorithm requires little time. In addition we consider G_1^s and G_2^s, a pair of non-isomorphic graphs on which the algorithm is slow, i.e., graphs where our algorithm requires a lot of time to terminate. (We also suppose that all four graphs are not isomorphic.) We now consider as input the disjoint unions $G_1 = G_1^f \cup G_1^s$ and $G_2 = G_2^f \cup G_2^s$. It is advantageous for our algorithm to analyze G_1^f (respectively G_2^f), instead of trying to understand the structure of G_1^s (respectively G_2^s). Thus, given this information, we prefer to choose for the pattern first all vertices from the graph G_1^f and the vertices from G_1^s.

In the absence of an optimal decision rule we use the following: We choose unique vertices, whenever possible, i.e., if a vertex has a unique adjacency structure to $p_1, \ldots, p_{t-1}$, i.e, to the vertices that have already been chosen, it should be chosen as p_t next. This way the sampling collects obvious information very cheaply and can determine erring choices faster. If there is no vertex with unique adjacency structure, we pick a vertex from the largest color class that remains, after a refinement of the vertex colors according to the adjacency structures. Another option is to use Nauty's individualization strategy. When using it we pick the vertex, whose individualization yields a refinement that partitions the colors class as much as possible. (Since in practice there is no need to generate the whole pattern in advance, it is produced on demand. Whenever the length of a sample would exceed the length pattern we have generated so far, the pattern is extended.) For a fixed screw box, the pattern is only required to calibrate the screws, i.e., to obtain their evaluation on the pattern, to which evaluations of sample vertices are compared. Otherwise, if new screws are inserted to the screw box, we have to use the pattern to calibrate the new screws.

We continue our discussion of the options of the algorithm with the design of suitable statistical tests.

2.7 Advanced statistical tests for equal distribution

In the previous section we have studied several versions of the ScrewBox algorithm. They all share the property that they need a probabilistic way to ensure certainty on non-isomorphism. In other words, some statistical test has to be performed in order to gain certainty on non-isomorphism. In some sense randomized algorithms for decision problems and statistical tests are the same concept. They are procedures that describe how to perform experiments that provide answers which are true with a certain error probability. They do, however, differ in their terminology. When possible, we abstract our concrete problem within the ScrewBox algorithm to a statistical problem, in which case we also use the adequate terminology.

In the ScrewBox algorithm, we perform samplings in two graphs, and gather the termination lengths of the samplings that are performed. Thus when we sample in one of the inputs graphs we obtain an integer. For either graph the outcomes appear according to some distribution. These two distributions are equal if and only if the graphs are isomorphic.

Abstractly, the problem we face is the following: For $n \in \mathbb{N}$, given two random variables $h_1, h_2 \colon \Omega \to \{0, \ldots, n\}$, we want to statistically infer that h_1 and h_2 are not equally distributed. Thus we are not primarily interested in verifying that the two variables are equally distributed, which is the case if the input graphs are isomorphic, see Theorem 8. If they are (and our test does not wrongfully show them to be unequally distributed), some isomorphism is produced eventually. We therefore formulate as null hypothesis the equality of the distributions, against which we want to test:

$$\begin{array}{llll} \mathcal{H}_0\colon & h_1 \text{ and } h_2 \text{ are} & & \text{equally distributed.} \\ \mathcal{H}_1\colon & h_1 \text{ and } h_2 \text{ are} & \textit{not} & \text{equally distributed.} \end{array}$$

Our goal is to minimize the number of experiments performed, as it corresponds closely to the running time of our algorithm (one essentially depends linearly on the other). The number of tests that are performed is not fixed in advance and it should be chosen in a sensible manner. When testing for equal distribution of h_1 and h_2, we additionally face the problem that the distributions of h_1 and h_2 are unknown, and complicating things even more, by changing the setup of the screw box, we may drastically alter these distributions.

Summarizing, we are interested in a statistical test with the following properties:

1. It is applicable, irrespective of the distribution of h_1 and of h_2.
2. It wrongfully rejects $\mathcal{H}_0$ with error probability of at most α. (The error of the first kind).
3. It wrongfully accepts $\mathcal{H}_0$ with error probability of $\beta = 0$, (i.e., if $\mathcal{H}_1$ is true then the test result does not claim the opposite, the error of the second kind).

4. If $\mathcal{H}_0$ is true, (i.e., if h_1 and h_2 are equally distributed), then the test may (and should) perform infinitely many samples.

5. It is economical, (i.e., it performs as few samplings as possible).

If we employ the test within the ScrewBox algorithm, Property 4 of the test will guarantee us that, while the test in principle runs forever, at some point an isomorphism is found at which point we end the test, as we do not require the result of the test.

Before we treat the general case, in which we consider two variables h_1 and h_2 with values in the non-negative integers $\{0, 1, \ldots\}$, we first consider, as an instructive example, the restricted case, in which only two values (0 and 1) are attained by the variables.

2.7.1 Testing a biased coin

In a restricted scenario, we consider $h_1, h_2 \colon \Omega \to \{0, 1\}$, two random binary variables, which are to be tested against equality of their distributions. We repeatedly and independently evaluate h_1 and h_2 simultaneously, until the outcome of the variables is not the same for the first time and use the last outcomes to design a new random variable X:

$$X := \begin{cases} 1 & \text{if } h_1 = 0 \text{ and } h_2 = 1, \\ 0 & \text{if } h_1 = 1 \text{ and } h_2 = 0. \end{cases}$$

We observe: h_1 and h_2 are equally distributed if and only if $\mathbb{E}(X) = \frac{1}{2}$, i.e., the expected value of X is $\frac{1}{2}$. As a variable that attains only two values, we consider X as the outcome of a coin toss. Thus, we have reduced our problem to the problem of determining whether a coin is biased or not.

We design a test that rejects the hypothesis $\mathcal{H}_0 : \mathbb{E}(X) = \frac{1}{2}$, with a significance level of α say. Properties 1–5 from the beginning of this section now translate into the requirements that the test is applicable for any coin that, when given an unbiased coin, will run forever with probability of at least $1 - \alpha$, and that it will never claim that the coin is unbiased. Additionally the test is supposed to perform few samples.

We let $\varepsilon := |1/2 - \mathbb{E}(X)|$ be the bias of the coin. For a known potential bias ε, Wald's sequential probability ratio test [127] solves this problem to optimality. By "knowing the potential bias" we mean that we are guaranteed (by some other source) that either $\varepsilon = 0$ or that ε is at least as big as some fixed minimum deviation from the unbiased case. For the coin tossing problem, Wald's test provides, based on the number of coins tosses performed and the observed number of heads, a rule that determines the future course of action: Either one of the hypotheses $\mathcal{H}_0$ or $\mathcal{H}_1$ is to be accepted, or one has to continue testing. In our case the hypothesis $\mathcal{H}_0$ is never accepted. It can be shown [128] that with this technique the expected number of coin tosses is in $\Theta(1/\varepsilon^2)$, if the coin is biased, and that this is the best possible.

Since, however, the potential bias ε is unknown to us, we have to use a different technique. We first convince ourselves that our desired properties for the test may actually be achieved: We perform coin-flips, until we have an estimate of how biased the coin is. For example, we can estimate ε with the maximum likelihood method. Assuming the validity of this estimate, we test the coin against the hypothesis $\mathcal{H}_0$, with some simple test that produces an error of the first kind of at most $\alpha/2$. If the test rejects $\mathcal{H}_0$, we claim that $\mathcal{H}_0$ is false. If not we restart: We perform further coin flips to improve our estimate of the bias. We then again apply a test on $\mathcal{H}_0$, but this time we require an error probability of at most $\alpha/4$. This procedure (flipping coins to improve the estimate, followed by a test with error probabilities $\alpha/8, \alpha/16, \ldots$) is either continued indefinitely, or we eventually reject $\mathcal{H}_0$, with cumulative maximum error of the first kind of at most α. (When we apply this test in the ScrewBox algorithm, we terminate once an isomorphism has been found with the sampling process, and thus will almost surely not continue the test indefinitely.) The procedure just described already provides all our desired properties, except that it is not very economical.

Thus we now develop an improved, more economical version. The essential ideas for this improved version are taken from a paper of Karp and Kleinberg [69], in which they analyze binary search under uncertainty. We adapt their methods, hidden in Lemma 3.4 in [69], slightly to our problem. Beside these methods, we also need their corollary of a generalized version of Azuma's inequality:

Lemma 2. *If a coin has a probability $p \in [0,1]$ of showing heads, and X_n is the series of random variables that count the number of heads after n tosses, then for any $q := p - \varepsilon$, with $\varepsilon > 0$ we have:*

$$P(X_n \leq qn) < e^{-\varepsilon^2 n/2}.$$

Proof. The proof is contained in [69], it uses Azuma's inequality for submartingales. □

With this lemma we may design an economical test, that detects the bias of a coin:

Theorem 12. *We let $\alpha \in (0,1]$ be a fixed error probability. There is a test that proves (with error probability of at most α) that a given biased coin (which shows heads with probability $p \in [0,1]$) is indeed biased, which performs an expected number of coin tosses in $\mathcal{O}\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$.*

Here $\varepsilon := |p - \frac{1}{2}|$ is the bias of the coin unknown to the algorithm. Given an unbiased coin, the test will run indefinitely with probability of at least $1-\alpha$.

Proof. We first design a coin toss algorithm that achieves the running time as required (see Algorithm 3), and then prove the claimed bound on the expected number of tosses performed. The algorithm runs in phases $k = 1, 2, \ldots$. In phase k we set $\gamma_k := e^{-k}$ and we perform $n_k := \lceil 16\gamma_k^{-2} \ln((k+2)/\sqrt{\alpha}) \rceil$ coin tosses. If in phase k the number of outcomes heads is in between $\frac{(1-\gamma_k)}{2} n_k$ and $\frac{(1+\gamma_k)}{2} n_k$, we continue to the next phase, otherwise we claim, possibly erring, that the coin is biased. By Lemma 2, the probability that the algorithm errs in phase k, is bounded by

$$2e^{-\gamma_k^2 n_k/8} \leq 2e^{-2\ln((k+2)/\sqrt{\alpha})} = \frac{2 \cdot \alpha}{(k+2)^2}.$$

So, in total, the probability that the algorithms errs is bounded by $\sum_{k=1}^{\infty} \frac{2 \cdot \alpha}{(k+2)^2} \leq \alpha$.

To bound the expected number of tosses, we assume $\varepsilon \neq 0$ is the bias of the coin. We define $\ell := \lceil \ln(1/\varepsilon) \rceil$. This way $\gamma_k \leq \varepsilon$ holds in all phases $\ell, \ell+1, \ldots$. The algorithm only continues past phase $k \geq \ell$ if the number of observed heads deviates by more than $\varepsilon n_k/2$ from the expected value. The probability of this happening is, by Lemma 2, bounded above by

$$2e^{-\varepsilon^2 n_k/8} \leq 2e^{-2\varepsilon^2 \exp(2k) \ln((k+2)/\sqrt{\alpha})} = 2\left(\frac{k+2}{\alpha}\right)^{-2\varepsilon^2 \exp(2k)}.$$

So, as Karp and Kleinberg argue: The number of tosses performed in phase k grows exponentially with k, while the probability of reaching phase $k \geq \ell$ decreases faster than exponential. Therefore the expected number of tosses is dominated by the running time of phase ℓ. More formally, if X is the random variable that counts the number of tosses, then

$$\mathbb{E}(X) \leq \sum_{k=1}^{\ell} n_k + \sum_{k=\ell+1}^{\infty} n_k \cdot 2\left(\frac{k+2}{\alpha}\right)^{-2\varepsilon^2 \exp(2k)} \in \mathcal{O}\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right).$$

This shows the desired bound on the expected number of tosses. $\square$

We have seen in the previous theorem that with $\mathcal{O}\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$ tosses we can determine whether a given coin is biased, where ε is the potential unknown bias of the coin. To show that this is optimal, we use the following lower bound on a related coin toss problem shown in [69]:

Theorem 13 (lower bound for direction of a bias [Karp, Kleinberg [69] (2007)]). *We let $\alpha \in [0,1)$ be a fixed error probability. Assume we are given a coin, known to be biased, but the bias $\varepsilon \in (0, 1/2]$ is unknown to us. There is no algorithm that determines with $o\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$ tosses whether the probability of heads p satisfies $p > 1/2$ or $p < 1/2$.*

We now use this lower bound to obtain a lower bound for our problem: We reduce the problem of determining the direction of a bias to the problem of determining the existence of a bias. For this reduction, we describe how any algorithm A that asserts that a coin is biased may be used to create an algorithm A' which determines the *direction* of the bias, and which only uses as many coin flips as performed by algorithm A.

Corollary 2 (lower bound for asserting the bias of a coin). *We let $\alpha \in [0,1)$ be a fixed error probability. There is no test that proves (with error probability of at most α) that a given biased coin, that shows heads with probability $p \in [0,1] \setminus \{\frac{1}{2}\}$, is indeed biased, when at the same time the expected number of coin tosses is in $o\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$. Here $\varepsilon := |p - \frac{1}{2}|$ is the bias of the coin unknown to the algorithm. The error probability of α requires that for unbiased coins the algorithm may wrongfully declare the coin biased with probability of at most α.*

Proof. Assume we are given an algorithm A for the problem of asserting bias that performs an expected number of coin tosses in $o\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$. For a given coin, we determine the direction of the bias in the following way: We run the algorithm, until it concludes that the coin is biased. We then claim that the coin is biased towards the direction of the result which occurred more often. (We break ties arbitrarily.) If algorithm A does not conclude the coin to be biased, we do not claim any direction of the bias. This procedure yields a new algorithm A' that outputs the direction of the bias.

By Theorem 13 it suffices to show that the probability of error of algorithm A' is bounded by α.

Case 1: We first assume that the coin is unbiased. In this case the algorithm A errs with probability of at most α, so the probability that algorithm A' errs is also bounded by α.

Case 2: It remains to bound the error probability of algorithm A', when the coin is biased. W.l.o.g., we assume that the coin is biased towards heads. We let $\mathcal{E}$ be the event that algorithm A claims that the coin is biased and that during the execution tails can be observed at least as often as heads. Since $\mathcal{E}$ is exactly the event in which algorithm A determines that the coin is biased, but algorithm A' fails to determine the correct direction of the bias, it suffices now to show that this event occurs with probability of at most α, (i.e., to show that $\mathcal{P}(\mathcal{E}) \leq \alpha$).

We now show that the probability of the event $\mathcal{E}$ does not decrease when algorithm A' (or equivalently algorithm A) is supplied with an unbiased coin, as opposed to a coin biased towards heads. Consider a specific outcome ω of the event $\mathcal{E}$, i.e, a sequence of observations of coin tosses with at least as many tails as heads. Since the specific outcome ω has more outcomes with tails than with heads, the probability of outcome ω does not decrease when the experiment is performed with a fair coin, instead of a coin biased towards heads. Since this holds for all outcomes in $\mathcal{E}$, the total probability of event $\mathcal{E}$ when algorithm A is executed on an unbiased coin is at least as big as the probability of $\mathcal{E}$ when algorithm A is executed on the coin with bias towards heads. Since algorithm A may not err with probability larger than α on unbiased coins, we therefore conclude $\mathcal{P}(\mathcal{E}) \leq \alpha$. □

The running time achieved by Algorithm 3 is thus optimal. Karp and Kleinberg use the Kullback-Leibler [77] divergence (also known as relative entropy) to show Theorem 13. Possibly the Kullback-Leibler divergence can be used to directly show Corollary 2. See [30] for an introduction to information theory, including the concept of entropy.

Since we use Boole's inequality (i.e., a union bound) for the error-estimation, we may actually perform the different phases at the same time: In this case we do not reset the observations, and in phase k we perform additional coin tosses, so that the total number of tosses performed is equal to n_k.

Now that we know how to solve the restricted scenario, where our two random variables h_1 and h_2 only attain values in $\{0, 1\}$, we return to the general problem.

Algorithm 3 Testing for bias of a coin

Input: A coin to test and a significance level α.
Output: **Yes** if the coin is biased, or with probability of at most α if it is not biased. The test runs forever otherwise.

```
1: for $k = 1$ to $\infty$ do
2:   reset all observations
3:   $\gamma_k \leftarrow e^{-k}$
4:   perform $n_k = \lceil 16\gamma_k^{-2} \ln((k+2)/\sqrt{\alpha}) \rceil$ coin tosses
5:   if the number of heads observed is more than $\frac{(1+\gamma_k)}{2} n_k$ or less than $\frac{(1-\gamma_k)}{2} n_k$ then
6:     return Yes
7:   end if
8: end for
```

2.7.2 Testing two random variables for equal distribution

In the previous subsection we have considered a simplified situtation in which the random variables h_1 and h_2 have two possible outcomes. This simplified situation, however, does not apply to the variables that model the data we collect when we use the ScrewBox algorithm.

The data we collect during the sampling process consists of the termination lengths T of the samples. (In practice we collect more detailed information to increase the statistical significance. We number the screws in the order they are applied and for every sample run note the number of the latest screw that was applied, but we ignore this detail and use only the length T.) Abstractly we are in the situation that we want to test whether two random variables $h_1, h_2 \colon \Omega \to \{0, \ldots, n\}$ have equal distribution.

We repeatedly evaluate h_1 and h_2 and gather the obtained data in a histogram. A *histogram* of length n is a map $H \colon \{0, \ldots, n\} \times \{1, 2\} \to \mathbb{N}$. For $j \in \{1, 2\}$, we denote by $H_j(t)$ the value $H(t, j)$.

The empty histogram is the all-zero function. A single sample run on graph G_j, with $j \in \{1, 2\}$, terminating with length T corresponds to the histogram with $H_j(T) = 1$ and 0 elsewhere. Samplings are performed in either graph G_j, according to the setup of the screw box, (see Figure 2.12).

The histogram of a set of sample runs is the sum of the histograms of all samples in the set, (i.e., the histograms are added as functions, by adding the function values). In particular, extending the sample set by another single sample run increases exactly one entry in the histogram of the entire sample set.

If after an equal number of samples from both graphs, G_1 and G_2, the values $H_1(t)$ and $H_2(t)$ differ significantly for some $t \in \{0, \ldots, n\}$, we have statistical evidence that the graphs are not isomorphic. If there is a particular length on which the histograms deviate extremely, we may use the concepts developed in the previous subsection. Note, however, that after samples have been performed, we cannot point to the length that shows the most significant behavior and assert conclusions with results from samples that have been performed in the past. Either we have to declare in advance which length t we consider, or we have to take account of the fact that the possible choices concerning this length t lower the significance of the result.

Since the deviation on each individual length of the histogram is usually not significant enough to claim non-isomorphism of the graphs with sufficient confidence, we develop a technique that increases the confidence by using entries of the histogram at multiple lengths.

If the two random variables h_1 and h_2 were fixed throughout the whole algorithm, we could apply a χ^2 test to show that they do not have equal distributions. We are, however, in a different situation: The distributions of the random variables change over time. The only fact we know for certain is, that at any given time, if we take a sample from each graph, (by applying the same screw box), the outcomes are equally distributed if and only if the graphs are isomorphic. In other words, the distributions of the random variables h_1 and h_2 change over time, but whether these distributions are equal remains fixed. We filter the histograms to cope with this situation and increase the significance:

Definition 24 (filter). A *filter*, specified by the coefficients $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_n) \in \{-1, 0, +1\}^n$, is a function F_σ that maps a histogram to a pair (a, b) of integers in the following way:

$$F_\sigma(H) := (a,b) = \sum_{t\colon \sigma_t = +1} \big(H_1(t), H_2(t)\big) + \sum_{t\colon \sigma_t = -1} \big(H_2(t), H_1(t)\big).$$

In words, the coefficients σ_t specify how the values of the histogram at length t contribute to $F_\sigma(H)$: by direct addition, by swapped addition, or not at all. Hence, $a+b$ is at most $\sum_t H_1(t) + \sum_t H_2(t)$, the total number of sample runs in either graph.

A good filter chooses the coefficients in such a way that the sampling process produces two integers with significant difference. In case the random variables h_1 and h_2 are equally distributed, (respectively the given graphs are isomorphic), the expected values of a and b coincide, irrespective of the filter we apply. Figure 2.13 depicts a good filter chosen from a histogram.

For $j \in \{1, 2\}$ we define the filtered outcome under variable h_j as $F_\sigma \circ h_j := \sigma_{h_j}$, i.e., the coefficient σ_{h_j} at the length equal to the value of h_j determines the value of $F_\sigma \circ h_j$. This way any filter also turns the random variables h_1, h_2 into filtered random variables $F_\sigma \circ h_1, F_\sigma \circ h_2$ with image in $\{-1, 0, 1\}$. We combine the variables $F_\sigma \circ h_1$ and $F_\sigma \circ h_2$ into one random variable:

$$F_{\sigma,h_1,h_2} := \operatorname{sgn}(F_\sigma \circ h_1 - F_\sigma \circ h_2)$$

with values in $\{-1, 0, 1\}$. Here sgn denotes the signum function. The expected value of the random variable F_{σ,h_1,h_2} is related to the difference of the distributions of h_1 and h_2:

Fact 2. *Let F_σ be any filter. If the two input graphs are isomorphic, then the function F_{σ,h_1,h_2} has an expected value 0.*

H_1	H_2
0	0
0	0
0	3
12	29
75	60
0	1
0	0
6	0

σ_t	H_1	H_2
	0	0
	0	0
1	0	3
1	12	29
	75	60
1	0	1
	0	0
-1	6	0

$F_\sigma(H)$	
12	39

Figure 2.13: The figure depicts a histogram H (left), a typical choice for a filter F_σ (middle), where $\sigma = (0, 0, 1, 1, 0, 1, 0, -1)$, given this histogram, and the result $F_\sigma(H)$ of the application of the filter to the histogram (right).

Conversely, if h_1 and h_2 are not equally distributed and we have not chosen the filter in an unfavorable way, then the expected value is different from 0. To determine whether the random variable X has expected value 0, we may ignore all outcomes of 0 of that variable: The expected value of X is 0 if and only if the expected value of X conditioned to $X \neq 0$ is 0. (Intuitively, when we try to determine whether a coin is biased, we may ignore all tosses for which the coin lands on the edge.) We thus ignore any evaluation with outcome 0 and have reduced the problem of determining whether two random variables $h_1, h_2 \colon \Omega \to \{0, \dots, n\}$ are equally distributed, to the problem of determining whether for one random variable $F_{\sigma,h_1,h_2} \colon \Omega' \to \{-1, 1\}$ the expected value satisfies $\mathbb{E}(F_{\sigma,h_1,h_2}) = 0$. We solved this problem (with range $\{0, 1\}$ and an expected value of $1/2$) in the previous subsection.

If the variables h_1 and h_2 are equally distributed, then for any choice σ, the coefficients of the filter, the filtered variable F_{σ,h_1,h_2} has expected value 0.

Let us rephrase this crucial observation: We may change the filter arbitrarily after having performed an equal number of samples in the graphs G_1 and G_2. Feeding the filtered result (possibly obtained with different filters) to Algorithm 3 yields a valid non-isomorphism test.

There are numerous choices for the coefficients of a filter, but they do not yield equally significant data. Thus, the question arises: How is a favorable filter determined?

2.7.3 Choosing an optimal filter

Intuitively we are interested in a filter that yields a filtered histogram which is unlikely to occur when an unbiased coin is tossed. A good filter F_σ thus provides a random variable, for which it is easy to statistically infer that it does not have an expected value of 0.

It is impossible to choose an optimal filter before we have sampled in the graph. We thus content ourselves with determining what the optimal filter is, given a histogram, i.e., given the data we have collected to far. Since there are exponentially many possible filters (for every σ_t, with $t \in \{0, \dots, n\}$, there are 3 choices, namely $+1$,-1 and 0). It is not obvious that a good filter can be determined efficiently. We first formalize what the quality of a filter is:

Let H be a histogram and F_σ a filter with a filtered histogram $F_\sigma(H) = (a, b)$. We define the *probability of the outcome H under filter F_σ* to be the probability that an unbiased coin shows at most a times tails when tossed $a + b$ times.

Given the filtered histogram $F_\sigma(H) = (a, b)$, the probability of an outcome H under a filter F_σ can easily be computed with the binomial cumulative distribution. It is defined as $\mathrm{bcd}(x; n, p) := P(X_n \leq x)$, the probability that the random variable X_n, that counts the number of tails in n tosses of a coin, which shows tails with probability p, evaluates to at most x. For the binomial cumulative distribution, our implementation uses a variant of the code that is contained in the Numerical Recipes in `C` [108].

Definition 25 (optimal filter). For a given histogram H an *optimal filter* is a filter under which the outcome of H is least probable. To avoid ambiguity we define for any optimal filter $\sigma_t = 0$, whenever $(H_1(t), H_2(t)) = (0, 0)$ for $t \in \{0, \dots, n\}$.

An optimal filter has very specific properties: Unless $H_1(t) = H_2(t)$, one possible choice for σ_t can be ruled out. The optimal filter accumulates all levels which disagree by at least a certain ratio, adds the smaller values at these levels to a number a, and adds the larger values to a second number b. Lengths t for which the outcomes $H_1(t)$ and $H_2(t)$ do not sufficiently disagree are ignored. We use the following lemma to argue this formally:

Lemma 3. *Let H be a histogram and F_σ, with $\sigma = (\sigma_0, \dots, \sigma_n)$, be a corresponding optimal filter with filtered histogram $F_\sigma(H) = (a, b)$. In this situation the following holds:*

1. $a \leq b$
2. *If for $t \in \{0, \dots, n\}$ we have $\sigma_t = 1$ then $H_1(t) \leq H_2(t)$.*
3. *If $H_2(t) = 0$ and $H_1(t) > 0$ then $\sigma_t = -1$.*
4. *If for $t, t' \in \{0, \dots, n\}$ we have $\frac{H_1(t)}{H_2(t)} \leq \frac{H_1(t')}{H_2(t')}$ and $\sigma'_t = 1$ then $\sigma_t = 1$.*

   (We assume $H_2(t)$ and $H_2(t')$ to be different from 0.)

Proof. We denote by $\mathrm{puc}(a, b)$ the probability that an unbiased coin shows at most a times tails and b times heads when tossed $a + b$ times, i.e.,

$$\mathrm{puc}(a, b) := \mathrm{bcd}(a; a + b, 1/2) = \sum_{i=0}^{a} \binom{i}{a+b} \cdot 2^{-(a+b)}.$$

1: If $a > b$ then the inverse filter $F_{-\sigma} = F_{(-\sigma_0,\ldots,-\sigma_n)}$ is strictly better than F_σ.

2: If $H_1(t) > H_2(t)$ and $\sigma_t = 1$ then we can improve the filter by changing σ_t to -1. This is due to the fact that if $a \leq b$ and $c > d$, then $\operatorname{puc}(a, b) \leq \operatorname{puc}(a + c, b + d)$.

3: This follows from the fact that $\operatorname{puc}(a + c, b) \geq \operatorname{puc}(a, b)$ and that $\operatorname{puc}(a, b) \leq \operatorname{puc}(a, b + c)$

4: This follows from the fact $\operatorname{puc}(a, b) \leq \operatorname{puc}(a+c, b+d)$ if $a \leq b$ and $a/b \geq c/d$. □

All statements in the lemma hold when $H_1(t)$ is interchanged with $H_2(t)$ and $\sigma_t = 1$ is interchanged with $\sigma_t = -1$ accordingly. Part 4 of the lemma tells us that for $t \in \{1, \ldots, n\}$ the highest value $r_t := \min\{\frac{H_1(t)}{H_2(t)}, \frac{H_2(t)}{H_1(t)}\}$, for which $\sigma_t \neq 0$, determines all coefficients of an optimal filter. We extend, strongly abusing notation, this definition to undefined fractions by setting $i/0 = \infty$, for these ratios, if $i \in \{1, 2, \ldots\}$.

Using the lemma we can efficiently construct an optimal filter. To find an optimal filter, we proceed in the following way: We order the lengths $t \in \{1, \ldots, n\}$ by their values of $r_t = \min\{\frac{H_1(t)}{H_2(t)}, \frac{H_2(t)}{H_1(t)}\}$. This ordering is given by $r_{t_1} \leq \ldots \leq r_{t_n}$, with $t_i \in \{1, \ldots, n\}$, say. Then for an optimal filter there is a cut-off ratio r such that $\sigma_{t_k} \neq 0$ if and only if $r_{t_k} \leq r$. In the example shown in Figure 2.13, this cut off ratio is $\frac{12}{29}$. After we have ordered the lengths according to the ratios, for every $i \in \{1, \ldots, n\}$ we evaluate the filter given by using the first i lengths, according to this ordering, (i.e., $t_1, \ldots, t_i$), and none of the lengths beyond that. For any length t, the orientation (whether $\sigma_t = 1$ or $\sigma_t = -1$) is determined by smaller number in the histogram at that length. Algorithm 4 summarizes how to determine an optimal filter in an efficient way.

If we assume that function calls to the binomial cumulative distribution bcd can be performed in constant time, we may bound the running time of the algorithm that computes an optimal filter. (The assumption is legitimate as for large parameters in practice we replace the binomial cumulative distribution by an approximation.)

Theorem 14 (running time for the optimal filter algorithm). *If we assume that calls to the binomial cumulative distribution can be performed in constant time, then the running time of Algorithm 4, the algorithm that determines an optimal filter for a given histogram H of length n, is in $\mathcal{O}(n \log n)$.*

Proof. The sorting step of Algorithm 4 can be performed in a time in $\mathcal{O}(n \log n)$. The remaining operations in Algorithm 4 are two loops iterated at most n times. All other operations can be performed in constant time. □

2.7.4 Testing with the ScrewBox

In Subsections 2.7.1–2.7.3 we have developed tools to perform efficient tests designed specifically for the situation we face. We now assemble the parts we have developed into the ScrewBox algorithm. The general framework for the algorithm is the one given by Algorithm 2.

The ScrewBox algorithm repeatedly chooses, using Algorithm 4, a filter, optimal according to the current histogram, and then performs a sampling in each input graph,

Algorithm 4 Determining an optimal filter

Input: Histogram H of length n

Output: An optimal filter F_σ, with $\sigma = (\sigma_0, \dots, \sigma_n)$.

```
 1: find an ordering bijection $\pi\colon \{0,\dots,n\} \to \{0,\dots,n\}$      // by sorting the values
    such that for $t < t'$ we have
        $\min\{\frac{H_1(\pi(t))}{H_2(\pi(t))}, \frac{H_2(\pi(t))}{H_1(\pi(t))}\} \le \min\{\frac{H_1(\pi(t'))}{H_2(\pi(t'))}, \frac{H_2(\pi(t'))}{H_1(\pi(t'))}\}$
                  // fractions with 0 in the denominator are consider as $\infty$, see text
 2: opt $\leftarrow$ 1, $i_{\text{opt}} \leftarrow 1$
 3: $(a, b) \leftarrow 0$
 4: for $i = 1$ to $n$ do
 5:   if $H_1(\pi(i)) \le H_2(\pi(i))$ then
 6:     $\sigma_i \leftarrow 1$
 7:     $(a, b) \leftarrow (a, b) + (H_1(\pi(i)), H_2(\pi(i)))$
 8:   else
 9:     $\sigma_i \leftarrow -1$
10:     $(a, b) \leftarrow (a, b) + (H_2(\pi(i)), H_1(\pi(i)))$
11:   end if
12:   if bcd$(a; a+b, 1/2) <$ opt then      // bcd: binomial cumulative distribution
13:     $i_{\text{opt}} \leftarrow i$
14:     opt = bcd$(a; a+b, 1/2)$
15:   end if
16: end for
17: for $i = i_{\text{opt}} + 1$ to $n$ do
18:   $\sigma_i \leftarrow 0$
19: end for
```

according to the rules in the screw box. This produces two numbers, T_1 and T_2, the termination lengths from each sampling. The filtered values are then used as the outcome of one coin flip in the statistical test given by Algorithm 3. Then the screw box is modified, according to Subsection 2.6.5, by inserting and deleting screws. This process is repeated until either the test asserts that the coin simulated by the filtered values is biased (i.e., the graphs are not isomorphic), or a sample of length n was found in graph G_2 (i.e., an isomorphism has been found).

If, as just described, we always use the filter, which is optimal given the current historgram, we face the following problem: During the execution, it is possible that the current filter yields significantly differing outcomes, but this significance disappears within the data we have collected so far. In this case it is preferable to reset the histogram and start a new test, without the noise previously collected. When we reset the data, we have to ensure that the error probability does not increase. One way to avoid this is to start the initial test with a probability of error of at most $\alpha/2$, where α

is the actual error bound we wish to guarantee. Resetting the data at any point in time and restarting the test with an error probability of at most $\alpha/2$, we ensure the required bound on the overall error.

Repeating this trick, it is possible to continuously start new tests with various filters, without violating the error bound: To perform multiple tests, we perform each of them with decreasing error probability, i.e., $\alpha/2, \alpha/4, \alpha/8, \ldots$, in order to maintain an overall error bound of α.

With this technique at hand, we propose an alternative method to the one given above which always supplies one test with the current optimal filter:

The alternative test applies various test filters, during a construction phase, in order to estimate the quality of the screw box. We consider this quality as "high" if the current histogram has a small probability under the optimal filter. Whenever the quality exceeds some prespecified bound (which becomes more stringent over time), we freeze the screw box (no screw insertions or deletions) and perform a test with it. If this test fails, we resume the modification phase of the screw box.

More explicitly, we proceed in three phases: In phase 1 we sample with the screw box, allowing modifications, until the probability of the histogram under the optimal filter is below a certain limit. We then fix in phase 2 the screw box and start a new histogram. We run samplings with the fixed screw box for some prespecified time and then choose an optimal filter. In phase 3 we perform a simple hypothesis test with this filter, i.e., we estimate the bias of the filter chosen in phase 2 and compute a number of samples N that are performed. If the histogram of the N samples deviates by more than a predetermined bound, which depends on the desired confidence level, we conclude that the input graphs are non-isomorphic. Otherwise we go back to phase 1.

The advantage of this type of test is that we obtain a randomized certificate for our computation, namely the current screw box and the optimal filter. Together they describe a set of sampling rules that prove the graphs to be non-isomorphic. We discuss this certificate further in Section 2.11. The disadvantage is that it requires more sampling runs to complete, a statement we will not quantify.

Apart from various technical details that have been omitted (see Section 2.9), this concludes our description of the ScrewBox algorithm. We now explain the advantages of this algorithm, and evaluate its performance. For this we first need to introduce further examples of graph constructions that yield challenging inputs for graph isomorphism algorithms.

2.8 Difficult graph instances

Throughout this document we frequently use the terminology "difficult graphs," upon which we elaborate in this section. We have already described two constructions that pose a challenge to graph isomorphism solvers: the CFI-graphs and the Miyazaki graphs, (see Section 2.4). As further examples, we now turn to strongly regular graphs and then to two types of graphs that arise from combinatorial constructions.

2.8.1 Strongly regular graphs

A strongly regular graph is a regular graph G for which the number of common neighbors of two distinct vertices $v, v' \in V(G)$ depends only on the adjacency of v and v' (i.e., whether v and v' are adjacent or non-adjacent). More formally:

Definition 26 (strongly regular). A graph $G = (V, E)$ is *strongly regular* if it is regular and there are non-negative integers λ and μ, such that any two distinct adjacent vertices $v, v' \in V$ have exactly λ common neighbors and any two distinct non-adjacent vertices $u, u' \in V$ have exactly μ common neighbors.

The strongly regular graphs can also be described as the graphs for which the 2-dimensional Weisfeiler-Lehman refinement refines the pairs of vertices v, v' into colors that only depend on the isomorphism type of the subgraph induced by $\{v, v'\}$, (i.e., the class of graphs isomorphic to the induced subgraph). As mentioned in Subsection 2.6.2, this also translates into a characterization via screws. For an introduction to strongly regular graphs see [25]. There are various generalizations of strongly regular graphs. A graph is said to be *t-tuple regular* if for any set $S \subseteq V$ of size at most t, the number of vertices that are adjacent to every vertex of S depends only on the isomorphism type of the graph induced by S. Cameron [24] shows, that any 5-tuple regular graph is t-tuple regular for any $t \in \mathbb{N}$. Furthermore the only graphs satisfying this property are disjoint unions of complete graphs of equal size, the 5-cycle and the line graph of $K_{3,3}$, the complete bipartite graph with 3 vertices in each partition. (The *line graph* of a graph $G = (V, E)$ is the graph $L(G) := (E, E')$, whose vertex set is the edge set of G, and whose edge set is defined, such that two vertices in $L(G)$ are adjacent, if they are incident as edges in G.)

Since for any tuple of vetices $(v_1, \ldots, v_t)$ of a graph G the value of $S^{t,1}(G, v_1, \ldots, v_t)$ in particular counts the number of vertices that are simultaneously adjacent to all v_i for $i \in \{1, \ldots, t\}$, we can translate the theorem on 5-tuple regular graphs into our terminology:

Theorem 15 (5-tuple regular graphs [Cameron [24](1980)]). *Let G be a graph. If for all $t \leq 5$ and all vertices $v_1, \ldots, v_t \in V(G)$ the value of $S^{t,1}(G, v_1, \ldots, v_t)$ depends only on the isomorphism type of the graph induced by $\{v_1, \ldots, v_t\}$ then G is $n \cdot K_r$, the n-fold disjoint union of complete graphs of the size r (for some $n, r \in \mathbb{N}$), the 5-cycle C_5 or the line graph $L(K_{3,3})$.*

In our framework we also require a notion of strong regularity that is applicable to colored graphs, and for which vertices of different color may behave differently. We say that a *colored graph is strongly regular*, if for any color c the number of c-colored neighbors of any vertex $v \in V$ depends only on the color of v. Additionally we require that for any color c the number of common c-colored neighbors of any two distinct vertices $v, v' \in V$ depends only on the colors and the adjacency of v and v'. Loosely speaking, we require that all properties which are invariant for a non-colored strongly regular graph, are invariant when the colors involved are equal. Analogously we may define t-tuple regularity for colored graphs.

We now consider two constructions that yield colored strongly regular graphs.

2.8.2 Hadamard matrices

In [91] McKay shows how Hadamard equivalence of Hadamard matrices can be solved via a reduction to Gi. The graphs obtained with this reduction are our first example of graphs that arise from combinatorial constructions.

Definition 27 (Hadamard matrix). An $n \times n$ *Hadamard matrix* is an $n \times n$ matrix A with entries in $\{-1, 1\}$ such that $AA^T = n\,\mathrm{Id}$.

The graph associated with an $n \times n$ Hadamard matrix $A = (a_{i,j})$, with $i, j \in \{1, \ldots, n\}$, is the graph with vertex set

$$V = \{v_1, \ldots, v_n, v'_1, \ldots, v'_n, w_1, \ldots, w_n, w'_1, \ldots, w'_n\},$$

and edge set E, such that (v_i, w_j) and (v'_i, w'_j) are edges in the graph, if $a_{i,j} = 1$, and (v_i, w'_j) and (v'_i, w_j) are edges in the graph if $a_{i,j} = -1$. Thus there are two vertices v_i, v'_i associated with every row i and two w_j, w'_j vertices associated with every column j. The row vertices are connected to the column vertices depending on the respective entry in the matrix A.

If we color the vertices corresponding to the columns with a different color than the vertices corresponding to the rows, the graphs associated with the Hadamard matrices are strongly regular (in the colored sense explained in the previous subsection). The second graph construction we consider builds on projective planes.

2.8.3 Projective planes

A *projective plane* of order N is an incidence structure on $N^2 + N + 1$ points, and equally many lines, (i.e., a triple (P, L, I) where P, L, I are disjoint sets, the points, the lines and the incidence relation with $I \subseteq P \times L$ and $|P| = |L| = N^2 + N + 1$), such that:

- for all pairs of distinct points $p, p' \in P$ there is exactly one line $\ell \in L$ such that $(p, \ell) \in I$ and $(p', \ell) \in I$,
- for all pairs of distinct lines $\ell, \ell' \in L$ there is exactly one point $p \in P$ such that $(p, \ell) \in I$ and $(p, \ell') \in I$,
- there are four points such that no line is incident with more than two of these points.

Figure 2.14 shows the only projective plane of order 2, the Fano plane. A famous open question asks whether there exists a projective plane of non-prime power order. In contrast to the fact that no projective planes of non-prime power order are known, there is an explicit construction for every prime power $N = p^k$: We obtain a projective plane of order N by considering the incidence relation of the 1- and 2-dimensional subspaces of $(\mathbb{F}_{p^k})^3$, the 3-dimensional vector space over the field of characteristic p that has p^k elements. A projective plane that arises by this construction is called

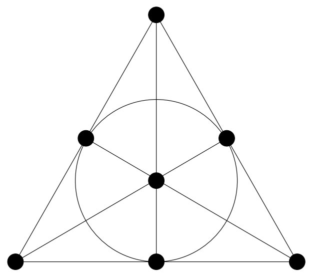

Figure 2.14: The Fano plane, the unique projective plane of order 2. The "line at infinity" is depicted as a circle.

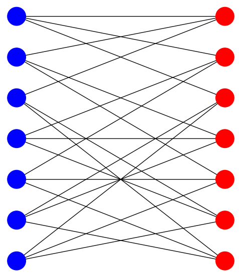

Figure 2.15: The incidence graph of the Fano plane.

algebraic. For various prime power orders non-algebraic planes are known. The web pages of Moorhouse [104] and Royle [113] offer a collection of known projective planes.

From a projective plane (P, L, I) we construct its incidence graph, a graph with vertex set $V = P \cup L$ and edge set $E = I$. Figure 2.15 shows the incidence graph of the Fano plane. Differentiating projective planes of the same order poses a difficult challenge to graph isomorphism programs.

As was the case with the colored Hadamard matrices, if we color the vertices originating from the points and lines with two different colors, then any projective plane is strongly regular.

Projective planes and Hadamard matrices are a special kind of combinatorial construction, they are combinatorial block-designs. For further information on the theory of block-designs, we refer the reader to the corresponding chapter in the Handbook of Combinatorics [18]. Deciding isomorphism of graphs that arise with the combinatorial constructions presented in this subsection poses challenging problems for graph isomorphism solvers. In subsection 2.10.2 we use them as benchmarks to evaluate the ScrewBox algorithm.

2.9 Engineering the ScrewBox

We have mainly taken a high level view to explain the ScrewBox algorithm. In this section we explain solutions to subproblems that arise during the implementation of the ScrewBox. In particular, we explain how to perform random samplings without replacement, and how to exploit sparse matrix multiplication, when the matrices are not given in sparse form. These two problems serve to illustrate the type of low level subroutines encountered when implementing the ScrewBox. They require algorithm engineering in order to guarantee short running times. Before we treat the matrix multiplication, we amend in Subsection 2.9.2 several details that were left out in the explanation of the ScrewBox in Section 2.6. In particular, we explain the preprocessing that is performed, which uses the matrix multiplication, and how the thereby obtained edge color information is used by the screws.

2.9.1 Random sampling without replacement

A core routine in the ScrewBox algorithm is the repeated sampling of vertices from a graph without replacement. We encode vertices by positive integers $\{1, \dots, n\}$, and sample from these integers. Frequently we do not sample a complete permutation but rather just a portion of these numbers. Then we start a new sample, i.e., we perform another sampling without replacement from $\{1, \dots, n\}$. In this subsection, we show how this can be done in constant time per sampling (apart from a one-time initialization).

We initialize an array a of size n by setting $a[i] = i$ for all entries $i \in \{1, \dots, n\}$ (i.e., the i-th entry of a is equal to i). The idea to guarantee constant time per sampling is the following: The entries of the array will always form a permutation of the integers $\{1, \dots, n\}$, and we maintain the numbers that have not been sampled in a consecutive initial part of the array, (i.e., we maintain them in the set $\{a[1], \dots, a[m]\}$ for some $m \leq n$).

To draw an integer that has not been drawn so far, we generate a random integer k in $\{1, \dots, m\}$. Our next element is $a[k]$. We then swap the value of $a[k]$ with the value $a[m]$, where m is the last position that contains an integer we have not drawn. Now the set $\{a[1], \dots, a[m-1]\}$ contains all numbers that have not been sampled so far. To draw the next integer we repeat the process with the value of m decremented. Algorithm 5 summarizes this procedure.

Algorithm 5 Random sampling without replacement

Input: The number of samples to be drawn $k \leq n$ and an array $a[i]$ of length n, such that the entries form a permutation of $\{1, \dots, n\}$
Output: A random sample $b_1, b_2, \dots, b_k$ of distinct integers in $\{1, \dots, n\}$

```
1: $m \leftarrow n$
2: $i \leftarrow 1$
3: while $i \leq k$ do
4:     draw $t$ uniformly at random from $\{1, \dots, m\}$
5:     $b_i \leftarrow a[t]$
6:     swap $a[t]$ with $a[m]$
7:     $m \leftarrow m - 1$
8:     $i \leftarrow i + 1$
9: end while
```

As desired, at any time during an execution of the algorithm, the entries in array a form a permutation of $\{1, \dots, n\}$. Thus, if the sampling is interrupted, i.e., no further elements will be drawn, there is no reason to reinitialize the array. A new sequence of samples may be drawn with help of the current values in the array.

Before we explain how we perform matrix multiplication, our next low-level implementation, we first describe where it is applied in the ScrewBox algorithm.

2.9.2 Pairlabel matrices

When approaching the isomorphism problem, it is reasonable to extract all information that can efficiently be gathered from the graph in a preprocessing step. In the preprocessing we attempt to differentiate vertices and refine the coloring, (as always without splitting orbits). For example, nodes with different degrees or nodes with different neighborhoods are separated this way. This is done by the naïve vertex refinement described in Section 2.2. We recall that any refinement splits color classes, but does not color vertices in the same orbit with different colors. In practice our algorithm colors the 2-tuples of the vertices. We call a matrix that has one entry for every ordered pair of vertices, and for which these entries are invariant under graph isomorphism, a *pairlabel matrix*. A coloring of the 2-tuples of vertices can be considered as a combination of an edge- and a vertex-coloring of the complete graph on n vertices. The adjacency matrix of a graph itself is a pairlabel matrix. Also the matrix that has as entries the colors given by the stable refinement of the 2-dimensional Weisfeiler-Lehman coloring procedure (see Definition 13), is a pairlabel matrix, since the procedure is invariant under graph isomorphism.

Instead of performing the computationally expensive 2-dimensional Weisfeiler-Lehman refinement we use the following strategy: We perform the naïve vertex refinement and update the entries on the diagonal of the pairlabel matrix. Once this refinement is stable for a pairlabel matrix A we perform a matrix multiplication step, with an algorithm described in the next subsection, and compute $kA + k'A^2$ for some fixed integers k and k'. If this does not refine the matrix A, we stop with the preprocessing. Otherwise, we go back to the naïve vertex refinement after which we use another matrix multiplication step. This is repeated, until the colors stabilize. The preprocessing is deterministic and invariant under graph isomorphism. When the ScrewBox is run on two inputgraphs, we first perform the preprocessing on each graph exactly the same way, i.e., we apply the same refinements in the same order. If the behavior of the graphs differs during this preprocessing, e.g., if the refined partitions are not of equal size, we conclude that the graphs are not isomorphic.

In addition to this preprocessing, we also use the pairlabels (i.e., the stable colors of the 2-tuples) during the sampling process. We do so by replacing the characteristic function λ that we used to define the screws in Definition 21 with the 2-tuple coloring obtained in the preprocessing, i.e., $\lambda(v, v')$ is always interpreted as a specific entry in the pairlabel matrix. The preprocessing is performed to enrich the information in the pairlabel matrix. This focuses the sampling process and saves computation time.

We remark that since the ScrewBox internally uses pairlabel matrices, i.e., 2-tuple vertex colorings of the graphs, the ScrewBox can also perform isomorphism tests on graphs which are edge- or vertex-colored.

2.9.3 Matrix multiplication

As explained in the previous subsection, we use matrix multiplication in the preprocessing step of the ScrewBox algorithm. This is our second example of a low-level implementation. Matrix multiplication is a 2-tuple coloring refinement procedure invariant under graph isomorphism: Given a 2-tuple colored graph $G = (V, E)$ of size n, we let A_G be the $n \times n$ matrix for which the entry $a_{i,j}$ is the color of the tuple (v_i, v_j). Reversing this association, the matrix A_G^2 may be considered as a new 2-tuple coloring. This new coloring does not separate orbits (i.e., for any two vertices v_i, v_j that lie in the same orbit, the colors (v_i, v_i) and (v_j, v_j) are equal): Any automorphism ϕ induces a permutation matrix P_ϕ. Since ϕ is an automorphism (and thus in particular preserves 2-tuple colors), we know that $P_\phi A_G P_\phi^{-1} = A_G$. Therefore $P_\phi A_G^2 P_\phi^{-1} = A_G^2$. The coloring is invariant under graph isomorphism: If we are given isomorphic graphs G_1 and G_2, in which we have chosen an ordering of the vertices, and an isomorphism ϕ' from G_1 to G_2, we again obtain a matrix $P_{\phi'}$, such that $P_{\phi'} A_{G_1} P_{\phi'}^{-1} = A_{G_2}$. From this we conclude $P_{\phi'} A_{G_1}^2 P_{\phi'}^{-1} = A_{G_2}^2$, thus, after refinement, the isomorphism ϕ' still preserves colors of 2-tuples.

When implementing the matrix multiplication, we refrain from using any complex data structures. We instead exploit the fact that the integers, as implemented in `C++`, form a (finite) ring $\mathbb{Z}_{2^B}$ (where B is the number of bits used per integer, which is machine-dependent). This means that we ignore the fact that integers overflow, since we know that this happens consistently with the ring operations.

There are two practical key observations that we use for our version of matrix multiplication:

1. It is desireable to have consecutive memory access since this minimizes the number of page faults, i.e., memory accesses that are not contained in the cache.

2. It is desireable to exploit the fact that the matrices that we multiply have a predominant entry c, i.e., almost all entries are equal to c.

We first assume that the matrices considered are sparse, i.e., that the predominant entry is 0. We will exploit the sparsity, even though the matrices are stored as adjacency matrix, rather than in a sparse format, by adjacency lists. By definition, the matrix product $AB = C$ (where the coefficients of the matrices are given by $a_{i,j}, b_{i,j}, c_{i,j}$ respectively) is computed by the formula $c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$. We can rewrite this as $c_{i,k'} = \sum_{k=1}^n \sum_{j=1}^n \delta_{k,j} \cdot a_{i,k} \cdot b_{j,k'}$, where the Kronecker-delta satisfies

$$\delta_{k,j} = \begin{cases} 1 & \text{if } k = j, \\ 0 & \text{otherwise.} \end{cases}$$

Since the coefficient $a_{i,k}$ is independent of j, we can transform the formula into

$$c_{i,k'} = \sum_{k=1}^{n} a_{i,k} \sum_{j=1}^{n} \delta_{k,j} \cdot b_{j,k'}.$$

We simultaneously compute the entries $c_{i,k'}$ by initializing them to 0, and then, iterating over i, k, j, we add to $c_{i,k}$ the product $a_{i,k} \cdot \delta_{k,k} b_{j,k}$. If $a_{i,k}$ is equal to 0, no

Algorithm 6 Matrix multiplication of sparse matrices with consecutive access

Input: Two Matrices $a_{i,k}, b_{j,k'}$ with $i,k,j,k' \in \{1,\dots,n\}$
Output: $c_{i,k'}$, with $i,k' \in \{1,\dots,n\}$ the product of the two matrices

```
1: initialize all $c_{i,k'}$ initialized as 0
2: for all $i \in \{1,\dots,n\}$ do
3:   for all $k \in \{1,\dots,n\}$ do
4:     if $a_{i,k} \neq 0$ then                // here sparsity saves computation time
5:       for all $k' \in \{1,\dots,n\}$ do
6:         $c_{i,k'} \leftarrow c_{i,k'} + a_{i,k} \cdot b_{k,k'}$
7:       end for
8:     end if
9:   end for
10: end for
```

iteration over j is required. This is exactly where we exploit the sparsity of the matrix. Algorithm 6 performs this iteration; it fulfills both desired properties mentioned above.

In practice, the matrices arising from the graph colorings are not sparse, in the sense that most entries are 0. However, there may be a value c, which is predominant.

In this case we can still use the effects of sparse matrix multiplication: We define $\mathbb{1}$ as the matrix for which every entry is equal to 1. We use the identity $A^2 = (A - c\mathbb{1})^2 + 2cA - nc^2\mathbb{1}$ to decompose the matrix multiplication into a sparse matrix multiplication followed by additions. (In practice we use $(A - c\,\mathrm{Id})^2$ as new coloring, since it contains the same information as A^2, unless some values are coincidentally hashed to equal values.)

This concludes our treatment of exemplary implementation details. With these details at hand we now perform an evaluation of the ScrewBox algorithm.

2.10 Evaluation of the ScrewBox algorithm

We first perform a theoretical comparison of the individualization refinement technique used by Nauty and the sampling approach used by the ScrewBox. Then we perform a practical comparison of running times of Nauty and the ScrewBox and finally analyze how the ScrewBox algorithm handles the CFI-construction.

A direct evaluation of the ScrewBox algorithm and the underlying sampling approach is difficult. While algorithms that perform the individualization refinement technique canonically label one input graph, the ScrewBox requires two input graphs, for which isomorphism is to be decided.

Many graphs, for which we perform experiments, do not come in pairs of graphs difficult to distinguish. For these graphs we run the ScrewBox on two isomorphic input graphs $G_1 = G_2$. This is no restriction: The time required to find an isomorphism of G_2, i.e., to find a sample of length n, bounds the time required to conclude non-isomorphism of two non-isomorphic graphs G_1 and G_2. Conversely however, for "difficult" graphs, the ScrewBox is more efficient on non-isomorphic graphs, even if these are very similar, and difficult to distinguish. To illustrate this ability of the ScrewBox, we analyze running times of the ScrewBox on pairs of non-isomorphic difficult graphs, such as projective planes, see Subsection 2.8.3. We start with a theoretical evaluation after which we present running times.

2.10.1 Theoretical evaluation

In this subsection we perform a theoretical evaluation of the ScrewBox algorithm.

An explicit bound on the expected running time of the ScrewBox algorithm is given by Theorem 9. It shows that on an input pair G_1, G_2 of graphs of size n the expected running time is in $\mathcal{O}(n^3 \cdot \frac{n!}{|\operatorname{Aut}(G_1)|} \cdot \log_2(1/\alpha))$, where α is the guaranteed bound on the probability of error. For the improved version, which uses k-level screws up to a level k', the corresponding bound amounts to $\mathcal{O}(n^{k'} n^2 \cdot \frac{n!}{|\operatorname{Aut}(G_1)|} \cdot \log_2(1/\alpha))$, which increases with increasing k'. As we cannot observe these bounds in practice, even on difficult inputs, they do not suffice to truly evaluate the sampling approach. To perform a theoretical evaluation we will therefore compare the ScrewBox to Nauty [92]. The intricacy in this is that Nauty is highly customizable with many options to choose from. In Subsection 2.6.4 we have presented ways to customize the ScrewBox. We have done this intentionally in a way, from which similarities between the sampling approach of the ScrewBox and the individualization-refinement approach of Nauty become apparent. This however enables us to perform a comparison of basic variants of the algorithms.

We compare a version of the ScrewBox that only considers a specific outcome of the sampling process to a version of Nauty that does not perform pruning with an indicator function. For either algorithm this is a major restriction. If we consider the algorithms in their full functionality, we cannot draw a theoretical comparison. Instead, for the unrestricted algorithms, we perform a practical comparison in the next subsection.

Nauty performs a backtracking search on a search tree, which depends on options with which it is run. When run with the same options the ScrewBox repeatedly samples a path from the root to a leaf in the same search tree as Nauty: For any choice of vertex invariants, (or vertex refinements) both algorithms obtain the same refined graphs, which represent the vertices of the search tree. Following [89], we call the vertices of the search tree nodes.

When sampling a path from the root in the search tree, the ScrewBox uses the screws to terminate the sampling. In order to prevent the ScrewBox from terminating a sampling before reaching a leaf node, we drop the choice of the pattern before the sampling process and resort to vertex refinements. We use a histogram that counts for every *type* of leaf node, (i.e., for every isomorphism type of a sample that could not be prolonged), the number of occurrences. If the ScrewBox visits a particular leaf a fixed number of times (depending on the significance level α), the algorithm terminates. In practice it is not possible to store information on all leaf nodes that have been visited. The same restriction is encountered by Nauty when it stores leaf vertices to perform automorphism pruning.

To compare the ScrewBox with Nauty, we only consider the runs of the ScrewBox on the input graph G_1 (and not those on G_2). If one leaf node s is reached a fixed number of times as dictated by some statistical test, the ScrewBox terminates: Either an isomorphism is found, because the ScrewBox has found a leaf node equivalent to s in graph G_2, or the ScrewBox terminates, concluding non-isomorphism, because leaf nodes of the type of s occur often in G_1 but never in G_2. (More formally: For a fixed ScrewBox, the expected running time of the ScrewBox for any input pair (G_1, G_2) is bounded by twice the expected running time on the input (G_1, G_1).)

We now relate the number of samplings performed by the ScrewBox to the number of tree leaves visited by Nauty. We first show a general lemma about the relation of the two basic search strategies in any search tree, which we now define.

Definition 28. Let T be a rooted tree where each non-leaf node is equipped with a probability distribution according to which the next child is chosen, i.e., edges are equipped with a probability p such that the edge probabilities of the children of a node sum up to 1.

- By the *backtracking algorithm BT* we denote the algorithm that recursively chooses a random ordering of the children of that node (possibly depending on the given distribution) and that proceeds with these children in the chosen order. (Basically BT models the "behavior of Nauty").
- By the *sampling algorithm SA* we denote the algorithm that starts at the root and repeatedly in every node chooses a child according to the given distribution until it reaches a leaf. It then restarts at the root. (Basically SA models the "behavior of the ScrewBox").

For any leaf s in the tree T we define $SA_T(s)$ as the expected number of leaves visited by the algorithm SA until it hits the leaf s. By BT_T we denote the number of leafs visited by the backtracking algorithm in tree T.

Lemma 4. *Let T be a rooted tree with ℓ leaves and internal nodes equipped with a probability distribution for their children. The algorithm BT visits every leaf of the tree, i.e, $BT_T = \ell$. Furthermore there is a leaf $s \in T$ such that:*

$$SA_T(s) \leq \ell,$$

i.e., the expected number of leaves visited by the sampling algorithm before visiting s is at most the number of leaves of the tree ℓ.

Proof. Algorithm BT searches the entire tree, therefore it also visits all leaves. The claim in the expected number of leaves visited by SA follows, if we choose s as the leaf that is visited most frequently. The sampling process ends in s with probability of at least $1/\ell$. Since the appropriate random variable is geometrically distributed, the expected number of samples performed until leaf s is reached is ℓ. □

The lemma relates repeated sampling from starting at the root and backtracking. The number of samples of the sampling algorithm is linearly bounded by the number of nodes visited by the backtracking algorithm. We can translate this statement into one that relates Nauty and the ScrewBox.

Theorem 16. *Assume Nauty and the ScrewBox are run with the same options (i.e., such that they traverse the same search tree), and ScrewBox is set to memorize all visited leaves. For any fixed significance level $\alpha \in (0,1]$, the expected number of samplings performed by the ScrewBox is bounded by a linear function in the number of search tree nodes that Nauty visits, when it does not prune the search tree with an indicator function.*

Proof. Let G be the input graph. (By the remark above, we only need to consider sampling runs of the ScrewBox on G_1.) Let T be the search tree of Nauty. The automorphism group $\operatorname{Aut}(G_1)$ acts on the levels (i.e., the sets of nodes with equal distance from the root), of the search tree. Consider $T/\operatorname{Aut}(G)$, the factor tree of T modulo this automorphism group. By performing automorphism pruning, Nauty avoids searching portions of the search tree. When it does not use an indicator function however, it always searches at least as many nodes as contained in the factor tree $|T/\operatorname{Aut}(G)|$. Each sampling projects onto the factor tree, since preimages of the tree are equivalent, it suffices for the sampling algorithm to visit some leaf in the factor a fixed number of times. We can therefore simulate the sampling algorithm on the factor graph $T/\operatorname{Aut}(G)$. The theorem now follows with Lemma 4. □

The theorem cannot straightforwardly be extended to a statement on running time as the ScrewBox repeatedly computes the same information for internal nodes on equivalent paths (i.e., paths that lie in the same orbit of the action of the automorphism group). If we assume that on every node roughly the same amount of work is performed, we can bound the running time linearly by the height of the search tree, which is at most n, and the number of samplings.

As Nauty and the ScrewBox become incomparable when equipped with further functionality, we can not compare by theoretical means: Nauty uses an indicator function to prune the search tree. Consequently, once certain parts of the tree have been visited, other parts will never be visited at all. The ScrewBox, on the other hand, is not bound to search for a specific leaf; it rather prunes the tree and simultaneously gathers information that is exploited in the tests. In particular, on difficult non-isomorphic input graphs, the ScrewBox will terminate before ever reaching a leaf.

The fact that the sampling approach can handle automorphisms is in accordance with the observation that CFI-graphs can be distinguished by the ScrewBox. (See Subsection 2.10.3).

2.10.2 Practical evaluation

The implementation of the ScrewBox is written in `C++`, without the use of special graph or matrix libraries, representing graphs as simple adjacency matrices. All tests have been performed on a 2.4 GHz AMD Opteron machine with one 1 GB RAM that runs Linux. The algorithm has been set to ensure an error probability of at most 0.05%.

Junttila and Kaski [65] have, in the course of designing the algorithm named Bliss, an engineered version of Nauty, collected a benchmark set of graphs on which they extensively tested Nauty and Bliss. We use this benchmark family to perform a practical comparison of the ScrewBox and Nauty. The graphs in this family consist of:

1. miscellaneous "easy" graphs (complete graphs, grid graphs, ...)
2. random regular graphs
3. the incidence graphs of algebraic affine and projective geometries
4. graphs arising from constraint satisfaction problems
5. graphs obtained by applying the Cai-Fürer-Immerman construction to random 3-regular graphs
6. Miyazaki graphs
7. random strongly regular graphs
8. graphs associated with Hadamard matrices
9. incidence graphs of algebraic and non-algebraic projective planes

As phrased in [65] these graphs were designed to evaluate 1) the efficiency of basic data structures, 2) heuristics for eliminating redundancy and 3) the efficiency of the implementation on truly large graphs.

The ScrewBox has not been implemented to solve "easy" instances. On these graphs the ScrewBox has cubic running time, as the preprocessing dominates the computation. This is reflected by running times for easy graphs (1, 2 and 3). This is also reflected by the running times on large graphs (4). In order for the ScrewBox to compete in such instances, a rigorous efficient implementation is required. As the individualization refinement technique, and in particular Nauty, have evolved in 30 years, this is a challenging task that has a different aim than the one taken in this thesis. Exemplarily Figure 2.16 depicts the running times on the grid graphs (contained in 1). These grid graphs are the Cartesian product of two paths.

The behavior of the ScrewBox on graphs obtained via the CFI-construction (5 and 6) is analyzed separately in Subsection 2.10.3. The random strongly regular graphs (7) are rigid, i.e., their automorphism group is trivial. The 2-dimensional Weisfeiler-Lehman refinement refines these into a coloring that induces a discrete partition. The same is true for the preprocessing that the ScrewBox algorithm performs. The running times on these graphs therefore do not yield meaningful results, since the time spent for the actual sampling is negligible.

We continue with running times for graphs associated with Hadamard matrices (8) and incidence graphs of projective planes (9).

To test the ScrewBox on graphs associated with Hadamard matrices, we used the family had-sw-44-$\langle i \rangle$ also contained in the benchmark family [65]. We chose this family as it contains large non-isomorphic graphs associated with Hadamard matrices of the same size. We ran the algorithm on pairs of these graphs. There is a large

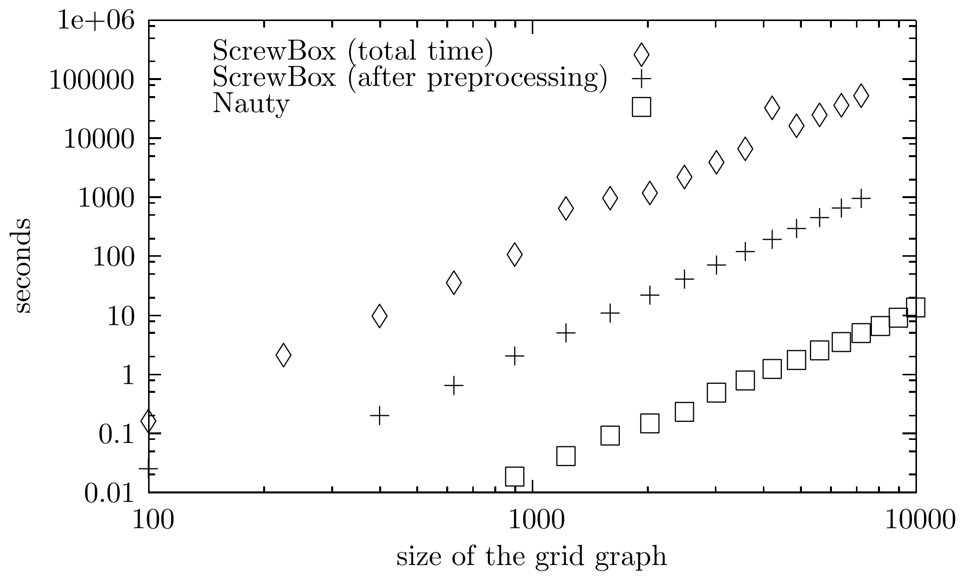

Figure 2.16: The figure depicts the running time of Nauty and the ScrewBox on grid graphs. It also depicts running time of the ScrewBox spent, after the preprocessing has been performed. Both axes are in logarithmic scale. The chart shows that in particular for "easy" such as the grid graphs, Nauty is very efficiently implemented. It also shows that both algorithm scale satisfactorily.

deviation on the single runs of the ScrewBox even on the same input pair. After 366 seconds 50% of the executions finished; after 3156 seconds 95% were finished. The longest run took 11798 seconds. With a simple doubling technique, i.e., restarting the algorithm after increasingly long intervals, the large deviation can be rectified and thus the outliers eliminated. In comparison Nauty required on average 93 seconds on these graphs. Thus, despite the very efficient implementation of Nauty, the ScrewBox achieves comparable running times on the Hadamard matrices.

In the original paper on the ScrewBox algorithm [79], an extensive test-suite on projective planes is performed. On these graphs a comparison between the ScrewBox and Nauty is conducted. Furthermore the projective planes are used as building blocks to devise larger instances of graphs that are "even harder." These larger instances are infeasible for Nauty. However the ScrewBox performs isomorphism tests on them without any tuning.

On projective planes, we performed only tests on pairs of non-isomorphic graphs with the ScrewBox. We split the graphs into several classes, on which the algorithms have similar running times.

Since the deterministic running times of Nauty vary only slightly within the considered graph classes, we simply list their averages. With the ScrewBox algorithm, we performed many runs on distinct pairs of graphs within the respective class. As with the graphs associated with Hadamard matrices, there is a large deviation among the running times even on the same pair of graphs. This is variation due to the choice of pattern. Therefore, we list the time it took 50% and 95% of the runs to complete. Though it is a randomized algorithm, all answers provided were correct. (The error bound we used 0.05% is only a crude upper bound of the actual error probability of the algorithm.)

		proj-16		proj-27		
		alg	n'alg	alg	n'alg	flag
Nauty	avg.	0 s	2 min	4 s	421 min	64 h
ScrewBox	50 %	2 s	2 min	18 s	39 min	73 h
	95 %	4 s	37 min	39 s	167 min	–

Figure 2.17: Running times for 21 runs the ScrewBox and Nauty on projective planes of order 16 and 27, ("proj-16" and "proj-27" respectively). Computations that involve algebraic planes ("alg"), and those that involve the planes "flag4" and "flag6" ("flag") are listed separately from the other computations with non-algebraic planes ("n'alg"). For Nauty the average running time is shown, if favorable for Nauty the "cellfano2" option has been used. For the ScrewBox the time after which 50% respectively 95% of the computations have finished is shown. (The Dash indicates that computations did not finish within three days).

Projective planes

We use all known projective planes of order $2^4 = 16$ and $3^3 = 27$, which can be found at the web pages of Moorhouse [104] and Royle [113]. There are 13 known planes of order 16 and 8 of order 27. (As geometric structures of points and lines, there are actually 22 respectively 13 known planes of these orders, but viewed as uncolored incidence graphs, planes cannot be distinguished from their duals, in which the points and lines are interchanged) We performed 21 ScrewBox runs on each pair of non-isomorphic planes of the same size.

For the planes of order 16 ("proj-16"), which have 546 vertices, the performance of our code is comparable to that of Nauty, while on the planes of order 27 ("proj-27"), with 1514 vertices, our algorithm was considerably faster than Nauty. The actual running times are depicted in Figure 2.17. The difficulty of the planes varies. For both, Nauty and the ScrewBox, algebraic planes ("alg") are much easier to solve than the non-algebraic ones ("n'alg"). Therefore all computations that involved algebraic planes are separated from the rest. Two exceptionally difficult incidence graphs of planes of order 27, called "flag4" and "flag6" on [104], are also listed separately.

Unions and joins

unions		1	2	3	4	6	8	10
Nauty	avg.	3	79	368	441	1101	2096	–
ScrewBox	50 %	1	2	4	7	13	19	32
	95 %	31	71	129	338	479	843	1403

joins		1	2	3	4	6	8	10
Nauty	avg.	1716	–	–	–	–	–	–
ScrewBox	50 %	1	2	4	6	13	25	34
	95 %	24	79	148	226	595	936	1277

Figure 2.18: Running times (in minutes) for the ScrewBox and Nauty of 21 runs on disjoint unions of projective planes $k \cdot P$ for $k \in \{1, 2, 3, 4, 6, 8, 10\}$, for four non-algebraic projective planes P of order 16 ("unions"), and these unions joined with an additional Fano plane $(k \cdot P) * F$ ("joins"). For Nauty the average running time is shown, if favorable for Nauty the "cellfano2" option has been used. For the ScrewBox the time after which 50% respectively 95% of the computations have finished is shown. (Dashes indicate that computations did not finish within three days).

In order to devise larger and more difficult graph instances, we combine several projective planes into one graph by forming disjoint unions and joins. By $r \cdot G$ we denote the disjoint union of r copies of the graph G, i.e., the graph

$$r \cdot G := \big(V \times \{1, \ldots, r\}, \big\{\{(v, i), (v', i)\} \mid (v, v') \in E(G), i \in \{1, \ldots, r\}\big\}\big)\,.$$

By $G * H$ we denote the *join* of graphs G and H, i.e., the graph

$$G * H := \big(V \cup V', E \cup E' \cup \{\{v_g, v_h\} \mid v_g \in G, v_h \in H\}\big)\,.$$

We ran the ScrewBox and Nauty on the unions $1 \cdot P, \ldots, 10 \cdot P$ and on the joins $(1 \cdot P)*F, \ldots, (10 \cdot P)*F$ for four non-algebraic projective planes P of order 16. Here F denotes the incidence graph of the Fano plane, depicted in Figure 2.15. Figure 2.18 shows the running times of the ScrewBox and Nauty on these graphs.

The ScrewBox proved to be very robust under the above graph operations. The running times range from a few seconds for the small instances to several minutes for a typical run on the large graphs. Combining several planes does not lead to an explosion of running times. In particular, joining an extra Fano plane to the disjoint unions only slightly increases the running time. This behavior is to be expected from the sampling strategy of the ScrewBox algorithm: The sampling tends to invest most of its resources in the "interesting" regions of the graph. An added Fano plane does thus not interfere with the discrimination of the base graphs P.

For Nauty, already the smallest instances of this collection are difficult. Large disjoint unions take several hours to compute. For one of the planes, the computation of Nauty on the 9-fold union did not finish within a week. Joining the Fano plane to the unions has a negative effect on Nauty's performance. The smallest graphs $(1 \cdot P) * F$ with non-algebraic P took several hours to compute and the $(2 \cdot P) * F$ cases did not finish within three days. In order to obtain canonical labelings, Nauty has to establish isomorphisms between all components. The extra Fano plane seems to complicate this task.

Nauty offers a number of options to adapt it to different classes of graphs. On each instance, Nauty has been executed with and without the "cellfano2" option, which is recommended for computation with projective planes. The tables only consider the faster run for each graph. For the unions and joins, Nauty does not benefit from the cellfano option. Presumably, it is possible to create a new invariant which helps in the recognition of this particular graph family.

2.10.3 The CFI-construction and the ScrewBox

In this subsection we return to the CFI-construction. Recall that the CFI-construction produces, from a connected base graph G, two non-isomorphic graphs $\mathrm{CFI}(G)$ and $\widetilde{\mathrm{CFI}}(G)$, by replacing the vertices of G with Fürer gadgets, and joining the gadgets with pairs of external edges. In the twisted replacement $\widetilde{\mathrm{CFI}}(G)$ in one of these pairs a twist has been introduced (see Section 2.3). We now analyze how the CFI-construction is handled by the sampling approach of the ScrewBox. Intuitively, it is irrelevant for the sampling process that many individualization steps have to take place in order to determine the graph structure of the CFI-graphs. For the sampling process, only individualization steps within a color class which is not refined into orbit poses a threat to the running time. When sampling in such a color class, the isomorphism type of the sample as a whole depends on the chosen vertex, and this possibly leads to an early termination of the sampling process. In the following, we argue with theoretical and practical arguments why the sampling algorithm is quite robust under the CFI-construction. For the following theorem recall that in our definition, when the CFI-construction is applied, the obtained graph is already a colored graph.

Theorem 17. *If a graph G does not have two 2-connected components which are connected (i.e., there is no path of bridges that connects two cycles), then the naïve vertex refinement refines the graphs $\mathrm{CFI}(G)$ and $\widetilde{\mathrm{CFI}}(G)$ into the orbit partition.*

Proof. It suffices to show that for any edge $e = \{v, v'\}$, the two associated outer vertices a_e^v, b_e^v are assigned different colors whenever a_e^v and b_e^v are not in the same orbit. We thus assume they are not in the same orbit. Then the original edge e in the graph G was not contained in a cycle. This means it is a bridge in G. Therefore in $G \setminus e := (V(G), E(G) \setminus \{e\})$ the two vertices v and v' are contained in two different connected components C and C' say. By assumption, one of the connected components C or C' does not contain a cycle. W.l.o.g we assume that this is the case for the connected component C' that contains v'. By induction on the number of vertices remaining in C' we conclude that for all edges $e' \neq e$ with $v' \in e'$, the outer vertices $a^{v'}_{e'}$ and $b^{v'}_{e'}$ are assigned different colors. Thus, in the Fürer gadget that replaces the vertex v', all but one pair of outer vertices are assigned different colors. In a Fürer gadget for every two inner vertices σ_1, σ_2 there are at least 4 outer vertices which are joined to exactly one of σ_1 and σ_2. Hence, all inner vertices of the Fürer gadget are colored with different colors. Therefore the last remaining pair $a^{v'}_e$ and $b^{v'}_e$, and thus also their neighbors a^v_e and b^v_e, are assigned different colors as well. □

If we individualize a vertex in a graph obtained via the CFI-construction, then apply the naïve vertex refinement and then delete singletons from the graph, we end up with a graph that is essentially a CFI-graph again. More precisely, we obtain a CFI-graph in which the inner vertices in a Fürer gadget are possibly duplicated. The obtained graph still refines into orbits if the respective underlying graph G', that is obtained from the original underlying graph G by deleting vertices, has no edge bridging two connected components.

Note that it is exactly this theorem that the Miyazaki construction circumvents, (as can be concluded from Figure 2.5). The Miyazaki graphs contain many bridges that connect cycles. In light of this observation, we tune our algorithm, when only faced with CFI-graphs, to use the vertex refinement as an option: Every time a vertex is added to the sample, we individualize this vertex and perform the naïve vertex refinement.

Figure 2.19 shows running times of the ScrewBox, without any adaption to the CFI-construction, on the CFI-graphs from the benchmark family devised by Junttila and Kaski [65]. These graphs are uncolored graphs that were obtained by applying the CFI-construction to random 3-regular graphs. The size of the graphs ranges from 200 to 2000.

The running time is dominated by the matrix multiplication steps performed in the preprocessing of the algorithm. (It is our version of the 2-dimensional Weisfeiler-Lehman vertex refinement and is explained in Subsection 2.9.2). This preprocessing performs $\mathcal{O}(\log(n))$ steps of matrix multiplication. It colors the original graph into its orbit partition. Figure 2.19 also shows the running times restricted to computation after the preprocessing.

We have argued that the naïve vertex refinement refines the CFI-construction whenever there is no bridge that connects two cycles. One may further show that the 2-dimensional Weisfeiler-Lehman refinement partitions the vertices of the Miyazaki graphs into orbits. (In fact, since they are of low pathwidth, this is not too surprising in the light of Theorem 4). The ScrewBox algorithm has therefore polynomial running time on these graphs, if we modify the options, such that every drawn sample vertex is individualized and the 2-dimensional Weisfeiler-Lehman refinement is

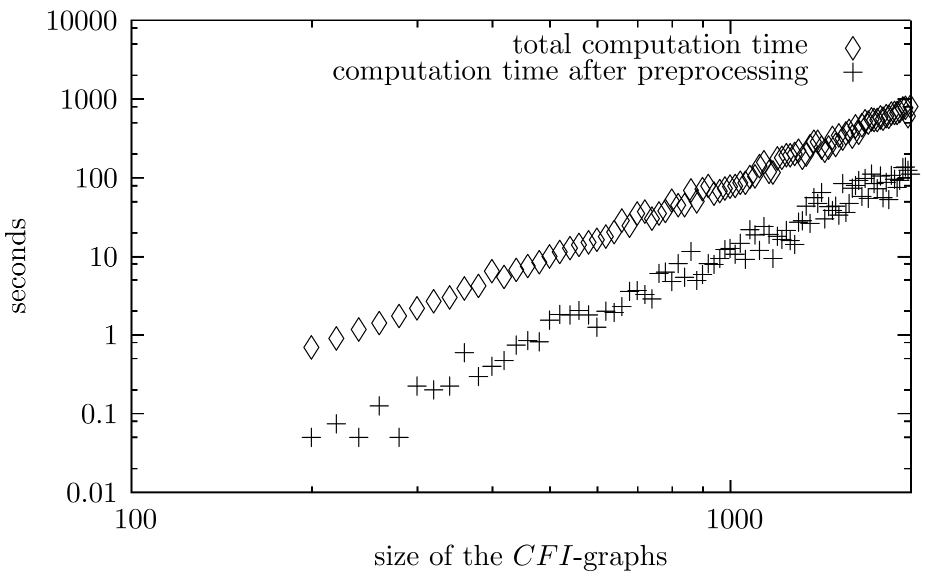

Figure 2.19: The figure depicts the running time of the ScrewBox algorithm on uncolored graphs that were obtained by application of the CFI-construction to random 3-regular graphs, taken from [65]. It also depicts running time spent after the preprocessing has been performed. The data in the log-log plot indicates polynomial running time of the ScrewBox on these graphs.

performed. Again, even without any modification, the ScrewBox algorithm mainly performs preprocessing on the Miyazaki graphs. Figure 2.20 shows the running times of the ScrewBox on the Miyazaki graphs from the benchmark family from [65]. This benchmark family also contains two families of Miyazaki graphs, which have been "reinforced with gadgets to mislead the cell selector." Nauty and Bliss have exponential running time on these graphs. As these gadgets are specifically designed for the cell selector of Nauty (and Bliss), it is not surprising that they do not affect the ScrewBox. The running times of the ScrewBox are almost identical on graphs with or without these reinforcements. Additional figures for these running times are therefore omitted.

We have shown in this subsection, theoretically and practically, that running times of the ScrewBox are robust under the CFI-construction. In contrast to the Weisfeiler-Lehman algorithm, the ScrewBox can handle graphs that require many individualizations: When chosen as next vertex for the sample, all vertices in the same orbit allow the sampling to continue exactly in the same manner.

This concludes the evaluation of the ScrewBox. In the next section we are concerned with the certification of the output of graph isomorphism algorithms in general and the ScrewBox in particular.

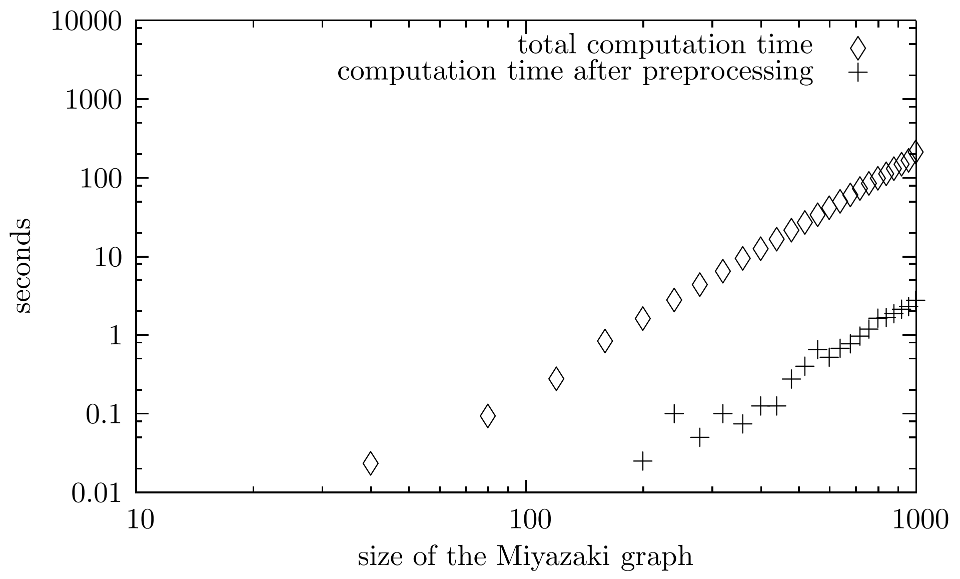

Figure 2.20: The figure depicts the running time of the ScrewBox algorithm on uncolored graphs obtained by application of the CFI-construction to the Miyazaki graphs, taken from [65]. It also depicts running time spent after the preprocessing has been performed. The plot shows that most of the running time is spent for preprocessing.

2.11 Certification

A standard graph isomorphism algorithm, given two input graphs, either returns an isomorphism between them or returns the claim "non-isomorphic." While in the former situation, the user can easily verify the correctness of the output by checking the isomorphism, in the latter situation, she is bound to trust the algorithm. This might not be very satisfying. The same difficulty occurs for any problem in $\mathcal{NP}$, not known to be in co-$\mathcal{NP}$: Generally speaking, a positive answer is easy to verify, while the correctness of a negative outcome can usually only be recognized through verification of the algorithm together with its computation.

The ScrewBox algorithm addresses this concern: When it terminates, claiming the input graphs are non-isomorphic, it has found with high probability a screw box that can be used to establish a difference between the two input graphs. In order to verify the correctness of the predicate, the user need not understand or even know anything about the construction process of the screw box. She only needs to convince herself that sampling with this particular screw box is invariant under graph isomorphism. The user employs the screw box to repeat a statistical test, with the filter that is also provided, with her desired error probability to confirm the non-isomorphism claim. In the next subsection we argue why this can be considered the randomized analogon to a certifying algorithm. For this, we first define a traditional (i.e., non-randomized) certifying algorithm. Mehlhorn and Näher [95] explain how certifying algorithms are used in the LEDA [96] library. See Mehlhorn's and Sanders' book on algorithms and data structures [97] for a current view on certifying algorithms.

Informally, a certifying algorithm is an algorithm that together with its output supplies a witness which certifies the correctness of the output. We require this witness to be "easy to check," where this non-mathematical term may mean various things: "Easy to check" may mean:

1. that a check can be performed quickly,
2. that it is easy to understand why the check really proves that the output is correct,
3. that the check has an easy implementation,
4. or any combination of the above.

We formalize these definitions now: Assume we are to design an algorithm that, given an input from a set X, is supposed to compute an output in the set Y. Here the user of the algorithm has to guarantee that a precondition $\Phi(x)$ for the input $x \in X$ is met. For every input $x \in X$ for which the precondition is met, the output $y \in Y$ of the algorithm must meet a postcondition $\Psi(x, y)$. Thus, the precondition is a map $\Phi\colon X \to \{\mathbf{true}, \mathbf{false}\}$, discriminating legal inputs, and the post-condition is a map $\Psi\colon X \times Y \to \{\mathbf{true}, \mathbf{false}\}$, discriminating legal outputs. The pair (Φ, Ψ) is called an *I/O-specification.* A randomized algorithm that errs with probability $\alpha \in [0, 1]$ computes, given an input that meets the precondition, with a probability of at least $1 - \alpha$ an output that meets the postcondition.

In the case of the graph isomorphism problem, the set X is the set of pairs of graphs. There is no precondition. (Alternatively, we can consider as precondition the requirement that the input must encode a pair of graphs). The output is either **Yes** or **No**. The postcondition $\Psi((G_1, G_2), y)$ is met if and only if $y \in \{\mathbf{Yes}, \mathbf{No}\}$ is the correct answer to the question: Are G_1 and G_2 isomorphic?

W.l.o.g., we assume that the element $\perp$ is not contained in Y and define the extended output set $\overline{Y} := Y \cup \{\perp\}$. A *strong witness predicate* for an *I/O-specification* (Φ, Ψ) is a predicate $\mathcal{W}\colon X \times \overline{Y} \times W \to \{\mathbf{true}, \mathbf{false}\}$, where W is some set of witnesses, such that for all $x \in X, y \in \overline{Y}, w \in W$ we have:

$$\big((y = \perp \wedge \mathcal{W}(x, y, w)) \Rightarrow \neg\Phi(x)\big) \bigwedge \big((y \in Y \wedge \mathcal{W}(x, y, w)) \Rightarrow \Psi(x, y)\big),$$

and which is additionally "easy to check."

I.e., if the output is $y = \perp$, then the witness proves that the input does not satisfy the precondition. Otherwise the witness proves that the input/output pair (x, y) satisfies the postcondition. If an algorithm provides a strong witness, this implies that it also determines the validity of at least one of two possibilities (whether the precondition is not met or the post condition is met). In contrast to this, we define a witness predicate:

A *witness predicate* for an *I/O-specification* (Φ, Ψ) is a predicate $\mathcal{W}\colon X \times \overline{Y} \times W \to \{\mathbf{true}, \mathbf{false}\}$ such that for all $x \in X, y \in \overline{Y}, w \in W$ we have:

$$\big((y = \bot \wedge \mathcal{W}(x,y,w)) \Rightarrow \neg\Phi(x)\big) \bigwedge \big((y \in Y \wedge \mathcal{W}(x,y,w)) \Rightarrow \neg\Phi(x) \vee \Psi(x,y)\big),$$

and which is additionally "easy to check."

Thus this weaker form of witness predicate only shows that the output is correct when the precondition is assumed. An algorithm that computes this kind of witness must not determine whether the precondition is not met or the post condition is met.

Definition 29 (strongly certifying algorithm). A *strongly certifying algorithm* for an I/O-specification (Φ, Ψ) is an algorithm for which a strong witness predicate $\mathcal{W}$ exists, such that the algorithm, given an input $x \in X$, computes a $y \in \overline{Y}$ and a $w \in W$ for which $\mathcal{W}(x,y,w)$ is true.

A *certifying algorithm* is defined analogously, by replacing the strong witness predicate with the ordinary witness predicate.

As indicated at the beginning of this section, we do not know how to construct a certifying algorithm for the graph isomorphism problem. A simple way to certify that two graphs are isomorphic is to provide an isomorphism, but no similar certificate is available for non-isomorphic graphs. For a decision problem a randomized algorithm is said to have a *one-sided error* if for one of the truth values **true** or **false** the algorithm always provides the correct answer. Randomized algorithms that err for both truth values are said to have a two-sided error. For a decision problem, we call an algorithm *one-sided certifying*, if for one of the truth values **true** or **false** the algorithm always provides a witness that certifies the answer.

The ScrewBox algorithm primarily tries to prove that two graphs are not isomorphic. Isomorphisms are merely produced as a side-effect. However, there is a generic way to turn a non-isomorphism test into an algorithm that finds isomorphisms. In fact any randomized colored graph isomorphism algorithm can be used to find isomorphisms. We will prove this statement by using a given non-isomorphism test as an oracle. For the proof we first require a standard lemma on error reduction of oracles:

Lemma 5. *Let Ocl be an oracle for a decision problem L, with fixed probability of error of at most $\varepsilon < 1/2$. For any $k \in \mathbb{R}$, we can simulate an oracle Ocl' for the problem L that errs with probability of at most $1/k$ by using $\mathcal{O}(\log(k))$ calls to the oracle Ocl for every simulated call to Ocl'.*

Proof. Given an oracle *Ocl* for problem L that errs with probability of at most ε, we can simulate the oracle Ocl' in the following way: Suppose we want to query Ocl' with the problem instance $Q \in L$. To simulate the oracle Ocl', we perform $\lceil \log_c(k) \rceil$ queries with problem instance Q to the original oracle *Ocl* (where c is a constant to be determined later). The majority of the answers of oracle *Ocl* is taken as the answer of Ocl'. We can bound the error probability of the new oracle with the Chernoff bound [59]. In particular we obtain that Ocl' gives a wrong answer with probability of at most $e^{-2\lceil \log_c(k) \rceil \cdot (1/2-\varepsilon)^2} = c'^{-\lceil \log_c(k) \rceil}$, where c' is a positive constant depending on ε. By choosing $c = c'$, we obtain $c'^{-\lceil \log_c(k) \rceil} \leq 1/k \in \mathcal{O}(\log(k))$. □

With the lemma we now prove, with the help of a self-reduction, that any randomized graph isomorphism algorithm can be made one-sided certifying with a one-sided error:

Theorem 18 (isomorphism certification). *Any randomized colored graph isomorphism algorithm A, possibly with two-sided error, can be turned into a one-sided certifying one-sided error graph isomorphism algorithm A'. If algorithm A has a running time in $\mathcal{O}(f(n))$, the certifying version A' has an expected running time in $\mathcal{O}\big(f(n) \cdot n^2 \cdot \log n\big)$.*

Proof. We suppose the original algorithm A has an error probability of $\varepsilon < 1/2$. We consider it as an oracle Ocl and use Lemma 5 to obtain a colored graph isomorphism oracle Ocl' that errs with a probability of at most $\frac{1}{3 \cdot (n^2+1)}$ and that performs $\mathcal{O}\big(\log(3 \cdot (n^2+1))\big) = \mathcal{O}(\log(n))$ calls to algorithm A for every query.

To show the theorem, we construct an algorithm A' that has an error probability of $\varepsilon' \leq 1/3$ and performs an expected number of $\mathcal{O}(n^2)$ calls to the new oracle Ocl'.

Given two graphs G_1 and G_2, the new algorithm A' first queries the oracle Ocl' as to whether the graphs are non-isomorphic. If the answer is "non-isomorphic," then the output of algorithm A' is also "non-isomorphic." If the oracle Ocl' claims that the graphs are isomorphic, the algorithm tries to find two permutations $v_1, \ldots, v_n$ and $v'_1, \ldots, v'_n$ of vertices in G_1 and G_2 respectively, such that the mapping that sends v_i to v'_i for all $i \in \{1, \ldots, n\}$ is an isomorphism. When initial parts of these sequences $v_1, \ldots, v_i$ and $v'_1, \ldots, v'_i$ have been found, an extension, by v_{i+1} and v'_{i+1}, can be found by the following procedure: We individualize one vertex v_{i+1} in the first graph, that is not contained in the sequence yet. (Recall that by Definition 12 an individualization of a vertex v assigns v a unique color). We consider a candidate for v'_{i+1} that is not contained in the sequence $v'_1, \ldots, v'_i$, and individualize it. We query the oracle Ocl' as to whether the colored graphs, in which $v_1, \ldots, v_{i+1}$ respectively $v'_1, \ldots, v'_{i+1}$ have been successively individualized, are isomorphic. For all candidates v'_{i+1} we perform this individualization and the isomorphism check. If a vertex v'_{i+1} can be found for which the graphs are isomorphic, then the sequences are extended by the respective vertices.

If the oracle Ocl' always gives the correct answer for all queries, and the graphs are isomorphic, the final sequences will induce an isomorphism between G_1 and G_2. Algorithm A' detects the case that Ocl' erred, i.e., that the final sequences do not form an isomorphism in the following way: The algorithm A' only claims that the graphs are isomorphic, if the sequences indeed represent an isomorphism. This can be checked in $\mathcal{O}(n^2)$ time. Therefore algorithm A' does not err when giving positive answers. By providing the isomorphism, it is also one-sided certifying. If the sequences obtained do not form an isomorphism, or at some point the answer of the oracle Ocl' assert that there is no possible extension of the sequences, we say that a failure has occurred. In this case we restart the whole procedure with another iteration, including the non-isomorphism check.

It remains to bound the probability of error of algorithm A' and to show that it has the required running time. By construction, the probability of error of Ocl' is at most $\frac{1}{3\cdot(n^2+1)}$. During an iteration of the algorithm, at most n^2+1 calls to oracle Ocl' are performed. The probability that all of the answers provided are correct is at least $1-\frac{n^2+1}{3\cdot(n^2+1)}=2/3$. In other words, the algorithm errs with a probability of at most $1/3$.

For the running time, we use the same bound. The probability that a failure occurs is bounded above by the probability that at least one answer in an iteration is not correct, which is at most $1/3$. The expected number of iterations E performed, until the first time that no failure occurs, is therefore at most $1+1/3+1/9+\ldots=3/2$. (Indeed if X_i is the random variable that indicates whether at least i iterations are performed, then $E=\mathbb{E}(\sum_{i=1}^{\infty}X_i)=\sum_{i=1}^{\infty}\mathbb{E}(X_i)=\sum_{i=1}^{\infty}(1/3)^{i-1}=3/2$.) Every iteration requires at most $\mathcal{O}(n^2)$ calls to the oracle Ocl', each of which calls $\mathcal{O}(\log(n))$ times algorithm A. As algorithm A has a running time of $f(n)$, in total we get an expected running time of at most $3/2\cdot\mathcal{O}(n^2)\mathcal{O}(\log(n))f(n)\subseteq\mathcal{O}(f(n)\cdot n^2\cdot\log n)$. □

The theorem shows that isomorphism can always be certified. As we do not know a way to certify non-isomorphism, we investigate alternatives to the deterministic version of certification.

2.11.1 Beyond deterministic certification

Theorem 18 deals with certification of isomorphisms. Concerning non-isomorphism, we do not know whether GI is in co-$\mathcal{NP}$, consequently we do not know how to succinctly, deterministically certify non-isomorphism. We claim that the ScrewBox is randomized certifying. Before stating precisely, what we mean by this, we make several observations that justify this terminology:

- Given two input graphs, the ScrewBox algorithm designs a screw box that behaves significantly different when it is used to sample in the two input graphs. This screw box and a filter are provided to the user, and serve as a witness.
- Using the given screw box, the user may resample and convince herself that the result is reproducible.
- She may use her own random source for this resampling.
- To understand the correctness of the algorithm, with certain guarantee on the error probability, it suffices to comprehend only a small fraction of the code. Given an encapsulation (which we define below) of all access to the input graphs from the remaining part of the code, it suffices to verify that screws are deterministic functions, with access to random vertices.
- The running time required for the verification is (much) shorter than the running time of the algorithm.
- The statistical test employed by the ScrewBox can be performed faster, after data on the distribution of the termination lengths of the sampling process in the two input graphs has been gathered.

When we refer to encapsulation of the access to a graph, we mean the following computational model: a deterministic Turing machine that instead of an input tape has access to an oracle that provides two features: 1) when requested, it supplies a random vertex $v \in V$ and 2) it answers queries that ask whether two vertices $v, v' \in V$ share an edge. For this type of query v and v' must be vertices that each are obtained via a request of form 1) to the oracle. The probability distribution of any deterministic function (with access to such an oracle) is invariant under graph isomorphism.

For a more formal definition of randomized certifiability, we alter the requirement "easy to check" from the beginning of Section 2.11 for witnesses supplied by a certifying algorithm. The check for the randomized version is allowed to be performed by a randomized algorithm. Thus, an algorithm is *randomized certifying* if together with its output it supplies a witness that may be used to statistically show, with a randomized algorithm, that the output is correct.

With the help of complexity theory, we describe how efficient the certification is. For this we define randomized certification complexity classes.

Definition 30 (randomized certification classes). Let $\mathcal{C}, \mathcal{C}_Y, \mathcal{C}_N$ be three complexity classes. We say that a decision problem L is in the certification class $(\mathcal{C} \mid \mathcal{C}_Y, \mathcal{C}_N)$ if there is an algorithm which, for any input $x \in X$, computes the output $y \in Y$ as either **Yes** or **No**, (whether x has the property to be computed or not) and supplies a witness $w \in W$. This algorithm must be in the complexity class $\mathcal{C}$. Additionally, for the witness predicate $\mathcal{W}$ the value of $\mathcal{W}(x, y, w)$ must be computable with a running time that is

$$\begin{array}{ll} \text{in } \mathcal{C}_Y & \text{if } y = \textbf{Yes}, \text{ and} \\ \text{in } \mathcal{C}_N & \text{if } y = \textbf{No}. \end{array}$$

In this definition we specifically allow non-deterministic complexity classes (in particular randomized ones).

As an example we consider the matrix multiplication decision problem, which, given three matrices $A, B, C \in \mathbb{R}^{n \times n}$, asks whether C is the product of A and B. This problem is in the class $(\mathcal{O}(n^{2.376}) \mid \mathcal{O}(n^2), \mathcal{O}(n))$: Coppersmith and Winograd [29] show that matrix multiplication can be performed in $\mathcal{O}(n^{2.376})$. If the product of the given matrices A and B is not equal to the putative product C, then a position of a specific entry, where AB and C differ, is used as witness. To check that the specific entry is different for AB and C, the user performs n multiplications and n summations, the negative check thus lies in $\mathcal{C}_N = \mathcal{O}(n)$. If on the other hand the product of A and B is equal to C, no witness is required. We can recheck that $AB = C$ by choosing random vectors $x \in \mathbb{R}^n$, and computing Cx and $A(B(x))$. This check can randomly be performed in $\mathcal{C}_Y = \mathcal{O}(n^2)$ time [71].

We remark, that this randomly certifying algorithm for matrix multiplication can also be used to certify the preprocessing of the ScrewBox algorithm explained in Subsection 2.9.2.

When using the notation to describe complexity classes we use a dot $(\cdot)$ to denote the complexity class of all functions (i.e., the class in which any algorithms lies). This way $\mathcal{NP}$ (respectively co-$\mathcal{NP}$) is the class of problems that are in $(\cdot \mid \mathcal{P}, \cdot)$ (respectively $(\cdot \mid \cdot, \mathcal{P})$).

We use the randomized certification classes to formulate a question concerning the graph isomorphism problem that has not been solved yet. As usual $\mathcal{BPP}$ is the class of randomized algorithms with bounded two-sided error.

Open Question 1. Is Gi, the graph isomorphism problem, in $(\cdot \mid \cdot, \mathcal{BPP})$? In other words, can non-isomorphism be certified with a witness which can then be verified in randomized polynomial time?

This question is a weaker form of the question whether Gi $\in$ co-$\mathcal{NP}$. In the language of Arthur-Merlin games Open Question 1 is exactly the famous open problem whether Gi is in $\mathcal{MA}$, see [72] for further detail.

The connection of the open question and the ScrewBox algorithm is the following: The ScrewBox algorithm provides a witness, i.e., a screw box, that can be randomly evaluated. The evaluation of the screw box heavily depends on this randomization. For difficult graphs however, we do not know how to construct a screw box that provides the non-isomorphism certification in randomized polynomial time. Thus, the question remains open.

The running time by which the randomized certification that employs the screw box as witness is (by far) shorter than the computation time required when the witness is not available. We do not know whether this can be expressed in the asymptotic notation. In the example of matrix multiplication given above, the randomized check for positive instances, which requires $\mathcal{O}(n^2)$ operations, does not require a witness. We do not have an example of an algorithm that provides a witness that is not deterministically checkable in a running time faster (in the $\mathcal{O}$-notation) than the one required for the computation of the witness.

In the next subsection, we consider a construction that makes the existence of such problems plausible. First, we give an example of a problem L that has a randomized algorithm that is faster than the best known deterministic algorithm that solves L. Given a graph G, the Min-Cut problem asks for a partition of the vertices into two parts that minimizes the number of edges, that have an endpoint in either part. Karger's randomized Min-Cut algorithm [67] solves the minimum cut problem in a running time of $\mathcal{O}(m \log^3 n)$ (and in $\mathcal{O}(n^2 \log n)$), while the best known deterministic running time of $\mathcal{O}(mn + n^2 \log n)$ is achieved by Stoer and Wagner's algorithm [122]. Karger and Panigrahi [68] recently showed it is possible to construct the cactus, a representation of all minimum cuts in the graph, also in near linear time. This algorithm is also randomized. We do not know how to enhance Karger's algorithm with additional output, without asymptotically increasing running time, such that we can (randomly) check the correctness of the output in a running time asymptotically shorter than that of Karger's randomized algorithm.

In the following subsection, we assume that L is a problem, for which the fastest algorithm is randomized. From this we construct a problem L^h that has a randomized certificate which may be checked fast. For the problem L^h all deterministic witnesses can be checked slower, unless it is possible to instances of L simultaneously.

2.11.2 Amplification of randomized certifiability

In this subsection we intuitively argue for the existence of problems that are randomly certifiable, but which lack equally efficient deterministic certificates. We do this with the help of a construction that amplifies the gap between the computation time required, when a certificate is available, and when it is not. For this we define the recursive majority of three: Let T be a rooted ternary tree of height h, i.e., a rooted tree in which every vertex is either a leaf or has exactly three children. This tree has 3^h leaves. To every leaf a truth value in $\{\textbf{true}, \textbf{false}\}$ is assigned. For any node in the tree we recursively define its truthvalue as the majority of the truthvalue of its children. The goal is to to evaluate the value at the root of the tree, but we wish to do so using as few values at leaves as possible. We can avoid having to inspect all truthvalues at the leaves with the following technique: We first evaluate two of the children of the root. If we are lucky and the values coincide, there is no need to evaluate the third subtree. Choosing the two subtrees uniformly at random, we are lucky with a probability of at least 1/3. Repeating this recursively yields an algorithm that for any input requires an expected number of evaluations of $(2 + 2/3)^h = (8/3)^h$ leaves. Jayram, Kumar and Sivakumar [64] show, that this bound can be improved to $(\frac{19\sqrt{1349}}{18})^h \approx 2.655^h$. In the same paper, they also show, using information theory, that no randomized algorithm can beat a lower bound of $(7/3)^h$. In contrast, when complete knowledge on the values at the leaves is available we may always choose 2^h leaves that certify the value at the root.

We now use the construction to obtain a problem that has randomized certificates. Consider a decision problem L, for which there exists a randomized algorithm that solves L, that is faster than any deterministic algorithm. (If the currently known algorithms for the MIN-CUT problem are optimal, then the MIN-CUT problem has this property). We form a ternary rooted tree of height h whose leaves correspond to an instance of the problem L. Again we define the value of a node as the majority of values of its children. Consider the new decision problem L^h that decides the value at the root. We may determine L^h with an expected number of 2.655^h of calls to an algorithm that solves the problem L. As before, there is a subtree with 2^h leaves that determines the value at the root. Given this tree, to check the solution it suffices for the user to recompute 2^h instances of the problem L by using the randomized algorithm for problem L. Since there is no algorithm that requires less than $(7/3)^h$ leaf evaluations in expectation, we have created an algorithm which may be checked with fewer evaluations than the expected number of evaluations required in the absence of a witness. In other words, we have shown the existence of a randomized algorithm that needs less evaluations when the certificate is known.

Be aware that this statement does not directly carry over to running time: Since a large input asks for the computation of many instances of problem L, it may be possible to determine the outcome by reusing partial information in the various instances: The instances may be in some relation to each other, in the worst case some of the leaf problems may actually be equal. If L were the graph isomorphism problem, for instance, a canonical labeling approach may circumvent the lower bound. One might resolve this issue by considering a restricted computation model, where results obtained in a leaf computation may not be used in the computation different leaf computation. As this would not be more expressive than the plausibility argument, we content ourselves with the latter.

Now that we have discussed the existence of problems that have a large gap between deterministic certification and randomized certification, we recapitulate that the ScrewBox algorithm is randomized certifying in three different ways:

1. It supplies a witness, which in practice may be checked considerably faster than the time required for the computation. The main reason is that during the modification phase of the ScrewBox, the data obtained by the sampling process becomes more and more significant. Moreover, the witness can only be checked by a randomized algorithm. Derandomizing the sampling process results in an exponential growth of the running time.

2. The preprocessing step of the ScrewBox uses matrix multiplication which can be certified with a randomized algorithm in a time faster than the running time of any known matrix multiplication algorithm.

3. The test applied by the ScrewBox can be performed faster when data has been collected, and the significance of the deviation may be estimates. For the restricted case of a biased coin we have even quantified this statement in Subsection 2.7.1. A known bias of ε can be asserted with a number of tosses in $\Theta(1/\varepsilon^2)$, where by Corollary 2 this cannot be done in $o\left(\frac{\log\log(1/\varepsilon)}{\varepsilon^2}\right)$, if the bias is unknown.

Summarizing, the ScrewBox is a practical example of an algorithm that has randomized certification.

2.12 Conclusion

Graph isomorphism remains one of the intriguing computational problems, whose complexity is not known. It is a representative of a class of problems that deals with combinatorial equivalence of relational structures, all of the same unknown complexity status. This chapter summarizes known algorithms that are used to approach the graph isomorphism problem, and develops a new randomized algorithm, the ScrewBox, that solves the general graph isomorphism problem. Many graph isomorphism algorithms have been engineered to quickly solve inputs that consist of graphs which are "easy." In contrast the ScrewBox aims at difficult graphs, for which isomorphism detection is infeasible for other algorithms, rather than at graphs for which the quality of an algorithm is measured by whether the answer can be found in seconds or milliseconds. The new sampling concept underlying the ScrewBox algorithm requires new theory and techniques. Both are thoroughly discussed in this chapter. The individualization refinement technique used by other algorithms such as Nauty, a practical graph isomorphism algorithm that has constantly been improved since it first appeared in

McKay's master thesis in 1976, is based on a backtracking search and automorphism pruning. The ScrewBox replaces this backtracking by repeatedly drawing random vertices. The ScrewBox also replaces group theoretical with statistical instruments. Though the ScrewBox is a practical algorithm, the aim of the chapter is to attract theoretical interest in alternatives to the classical approach taken to graph isomorphism.

Most of the chapter provides a high-level view of necessary theory to understand the ScrewBox algorithm, how it is made efficient, and why particular design choices are optimal. Exemplarily some low level implementation techniques have been outlined. Running times on various graphs have been provided to evaluate the ScrewBox algorithm and graphs. On a particular family the ScrewBox outperforms the benchmark isomorphism solver Nauty.

Certification of graph non-isomorphism is discussed, and alternatives for deterministic certifying algorithms are developed. The ScrewBox algorithm provides a practical way of certifying non-isomorphism. The certificate given by the algorithm, the screw bow with its optimal filter, can then randomly be checked.

When solving the graph isomorphism problem, the computation of a canonical labeling has the advantage that it allows to screen a graph against a large database. It is not apparent how to perform this screening with the ScrewBox algorithm. With the algorithm developed in Chapter 4 means to do so are supplied.

We close the chapter with a citation from Cai [22], who phrases the fact that the polynomially hierarchy collapses, if graph isomorphism is $\mathcal{NP}$-hard by saying: "It is *Not* likely that we can show that it is *Not* likely to be easy."

3 Van der Waerden numbers

In 1927 Bartel Leendert van der Waerden [126] was the first to prove Baudet's conjecture on arithmetic progressions within partitions of consecutive integers. He proved that whenever the integers are partitioned into finitely many parts, one of the parts contains an arithmetic progression of arbitrary length. Numerous generalizations, simplifications and variants of his proof have cumulated over time, forming the base for the Ramsey theory on the integers. These variants are as far reaching as Szemerédi's theorem [124], whose proof has supplied the mathematical community with priceless tools and insights into the natural numbers. The obtained tools and the acquired insight culminate in Green's and Tao's theorem [53], stating that the primes contain arbitrarily long arithmetic progressions.

The numbers that correspond to the original theorem by van der Waerden, accretively entitled van der Waerden numbers, quantify how many consecutive integers can be partitioned into a fixed number of parts, without the occurrence of an arithmetic progression of a certain length within one of the parts. For the van der Waerden numbers, there is a large gap between the known lower bounds, which are exponential, and the known upper bounds, which are given by a tower of twos (see Sections 3.2 and 3.3).

Rather than with asymptotic bounds, in this thesis we concern ourselves with the exact computation of such van der Waerden numbers. Computing them is, after the graph isomorphism problem, the second computational problem with unknown complexity status we investigate. More precisely, we deal with the more general problem of computing mixed van der Waerden numbers, for which the size of the arithmetic progression in consideration is allowed to vary among the parts. Upper bounds for the running times of contemporary algorithms depend on the value of the computed mixed van der Waerden number. In particular our available upper bounds on the running times range somewhere between linear and a tower of twos. This huge uncertainty in the running times exists despite the fact that the number of substructures of interest, namely the arithmetic progressions within a set of consecutive integers, is polynomially bounded in the size of the set. It also exists despite the fact that progressions of maximal length can easily be found in polynomial time, (see Section 3.5). (This is in contrast to the detection of cliques in a graph, with which we deal in the next chapter).

We commence with basic definitions in Section 3.1 and review available algorithms in Sections 3.6 and 3.7. In Section 3.8 we then introduce a new view of colored progressions, and, using this view, design an algorithm that computes mixed van der Waerden numbers. With the new wildcards algorithm, for the case where consecutive integers are partitioned into 2 parts, we verify all but one known mixed van der

Waerden number. For the case of at least three parts, our algorithm outperforms previously developed algorithms. For this case all previously known and two new mixed van der Waerden numbers are computed. The numbers are provided in Section 3.4.

3.1 Van der Waerden numbers

Ramsey theory in general deals with the necessity of the occurrence of certain substructures, when a larger structure is partitioned into finitely many parts. In this chapter we deal with the branch of Ramsey theory that is concerned with integers, more precisely we deal with colorings of integers. We commence with the necessary background essential to define the van der Waerden numbers. All of this can be found in Landman and Robertson's book addressing Ramsey theory on the integers [81]. For the sought van der Waerden numbers, the specific substructure is that of an arithmetic progression:

Definition 31 (arithmetic progression). A *k-term arithmetic progression with gap d* is a set of integers of the form $\{a, a+d, \dots, a+(k-1)\cdot d\}$ with $a \in \mathbb{Z}$ and $k, d \in \{1, 2, \dots\}$.

The positive integer k, the number of terms in a progression, is called the *length* of the progression. Two progressions *intersect* if they intersect as sets.

We say that two arithmetic progressions given by $\{a, a+d, \dots, a+(k-1)\cdot d\}$ and $\{b, b+d', \dots, b+(k'-1)\cdot d'\}$ *aim at the same term* if $a+k\cdot d = b+k'\cdot d'$, i.e., were both progressions extended by one additional term, they would be extended by the same integer. Figure 3.1 depicts these definitions.

Throughout this chapter we consider maps from subsets $S \subseteq \mathbb{N}$ of the non-negative integers to a finite set C. In analogy to the colored graphs (see Definition 2), we define such a map $\chi\colon S \to C$ as a *coloring* of the integers in S. We say the coloring has length $|S|$. The set C is called the set of *colors*, and is often given by $\{1, \dots, c\}$. Any coloring with $|C| = c$ colors is a *c-coloring*. Under a given coloring χ an arithmetic progression is *monochromatic* if the restriction of χ to the progression is constant, i.e., all terms of the progression have the same color. Figure 3.1 also depicts a 2-coloring of the integers with color set $\{0, 1\}$. One of the progressions shown is monochromatic.

Van der Waerden's theorem asserts that any coloring of the integers with finitely many colors induces monochromatic arithmetic progressions of arbitrary length.

An equivalent formulation of van der Waerden's theorem colors sets of consecutive integers (the fact that the two formulations are equivalent can be seen by an application of the compactness principle [51]):

Theorem 19 (van der Waerden's theorem [van der Waerden [126](1927)]). *For all positive integers $k, c \in \{1, 2, \dots\}$ there is a positive integer n such that any c-coloring $\chi\colon \{1, \dots, n\} \to \{1, \dots, c\}$ forms a k-term monochromatic arithmetic progression (with arbitrary gap).*

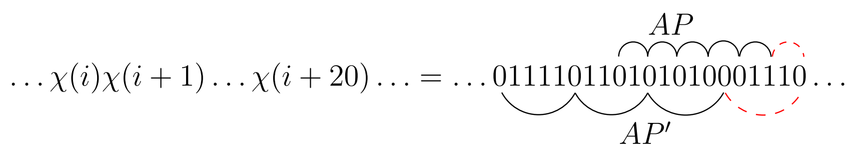
AP
$\dots\chi(i)\chi(i+1)\dots\chi(i+20)\dots = \dots 011110110101010001110\dots$
AP′

Figure 3.1: The figure shows parts of a coloring $\chi\colon \mathbb{N} \to \{0,1\}$. It also shows AP, a 6-term arithmetic progression with gap 2, and AP′ a 4-term *monochromatic* arithmetic progression with gap 5. The progressions intersect in term 2 of AP and term 3 of AP′. Both progressions aim at the same position, as indicated by the dashed lines.

In [50] Graham and Rothschild provide a concise proof of van der Waerden's theorem. In the next subsection, we describe a variant of their proof that provides a crude recursive upper bound.

The smallest number $w \leq n$, denoted $w(k;c)$, for which the conclusion of the theorem holds is called the *van der Waerden number* for c colors and progressions of length k. These numbers generalize to the case, in which the considered length k of the progression may vary with the color:

Definition 32 (mixed van der Waerden number). For any $c \in \{1, 2, \dots\}$ and any sequence $k_1, \dots, k_c$ of positive integers define the *mixed van der Waerden number* $w(k_1, \dots, k_c; c)$ to be the least integer w, for which any coloring of $\{1, \dots, w\}$ contains, for some $t \in \{1, \dots, c\}$, a monochromatic arithmetic progression of length k_t in color t.

The mixed van der Waerden numbers generalize the van der Waerden numbers, in particular $w(k;c) = w(k, \dots, k; c)$, (where k is repeated c times on the right side of the equation). We frequently refer to the mixed van der Waerden numbers simply as van der Waerden numbers. Since for any $k' \leq k_t$ any coloring that contains a monochromatic arithmetic progression of length k_t in color t also contains a monochromatic arithmetic progression of length k' in color t, for a fixed number of colors c the mixed van der Waerden numbers are monotone in every parameter k_t.

Given parameters c and $k_1, \dots, k_c$, we say that a c-coloring of a set of integers is *proper*, if for any color $t \in \{1, \dots, c\}$ it contains no monochromatic arithmetic progression of length k_t. Such a proper coloring is also called a $(k_1, \dots, k_c; c)$*-coloring*.

We now prove a first upper bound for the van der Waerden numbers.

3.1.1 Existence of van der Waerden numbers

In this subsection we develop a coarse recursive upper bound for the van der Waerden numbers, and thereby also proof their existence. The proof is essentially taken from the book on Ramsey theory by Graham, Rothschild and Spencer [51]. We turn this proof into a constructive version, that yields a doubly recursive bound. We generalize the van der Waerden numbers in order to facilitate the proof:

Definition 33. For $k, c, t \in \{1, 2, \dots\}$ let $B(k, c, t)$ be the minimum natural number, such that any c-coloring of $\{1, \dots, B(k, c, t)\}$ yields an arithmetic progression of length k or it yields t progressions of length $k-1$ monochromatic in t different colors, which aim at the same next term contained in $\{1,\dots,B(k,c,t)\}$.

We observe that $B(k,c,c)=w(k;c)$: By definition any coloring of $\{1,\dots,w(k;c)\}$ contains an arithmetic progression of length k, and therefore $B(k,c,c)\leq w(k;c)$. It remains to show that $B(k,c,c)\geq w(k;c)$. For this it suffices to show that any coloring of $\{1,\dots,B(k,c,c)\}$ contains an arithmetic progression of length k. By definition, such a coloring contains an arithmetic progression of length k or there are c progressions of length $k-1$ in c different colors that aim at the same position i. This position i is colored with j say, then there is a $(k-1)$-term progression in color j which, together with position i, forms a monochromatic arithmetic progression of length k. Therefore $B(k,c,c)\geq w(k;c)$.

For $t>c$ we observe that $B(k,c,t)=w(k;c)$: Indeed in case $t>c$ there cannot be t progressions all colored differently, thus the definitions of $B(k,c,t)$ and $w(k;c)$ coincide.

We now prove a recursive upper bound on the numbers $B(k,c,t)$:

Theorem 20 (van der Waerden recursion). *For the numbers $B(k,c,t)$ we get the following recursive bounds:*

$$\begin{aligned}
B(k,c,t) &\leq B(k,c^{B(k,c,t-1)},1)\cdot B(k,c,t-1) && (3.1)\\
B(k+1,c,1) &\leq \frac{k}{k-1}(B(k,c,c)-1)+1 && (3.2)\\
B(1,c,t) &= 1 && (3.3)\\
B(2,c,t) &= \min\{c,t\}+1 && (3.4)\\
B(k,c,t) &= B(k,c,c),\ \textit{if } t\geq c && (3.5)
\end{aligned}$$

Proof. We begin by proving Inequality 3.1: Assume χ is a c-coloring of the integer set $S=\{1,\dots,B(k,c^{B(k,c,t-1)},1)\cdot B(k,c,t-1)\}$. Divide the set S into $B(k,c^{B(k,c,t-1)},1)$ blocks $B_1,\dots,B_{B(k,c^{B(k,c,t-1)},1)}$ of $B(k,c,t-1)$ consecutive integers each. There are $c^{B(k,c,t-1)}$ ways to color each block with c colors. If one of these blocks contains a monochromatic progression of length k, we are done. Otherwise we consider the possible ways in which a block may be colored. We say two blocks are equally colored if for every $\ell\in\{1,\dots,B(k,c,t-1)\}$ the ℓ-th elements of both blocks have the same color. We have $B(k,c^{B(k,c,t-1)},1)$ blocks colored in $c^{B(k,c,t-1)}$ colors. By the definition of $B(k,c^{B(k,c,t-1)},1)$, there must be an arithmetic progression of length $k-1$ of equally colored blocks, $B_{i_1},\dots,B_{i_{k-1}}$ say. Since no block contains a monochromatic arithmetic progression of length k, in each of these blocks B_i there are $t-1$ colordistinct monochromatic progressions $p^i_1,\dots,p^i_{t-1}$ aiming at a term a_{B_i}. Since the blocks $B_{i_1},\dots,B_{i_{k-1}}$ are equally colored, we may chose these progressions with the same color and the same relative position in each block. Since the blocks $B_{i_1},\dots,B_{i_{k-1}}$ form an arithmetic progression, the terms $a_{B_{i_1}},\dots,a_{B_{i_{k-1}}}$ form a monochromatic progression of length $k-1$. This progression aims at some position a say.

We claim that there are t monochromatic progressions of different colors that aim at position a: For $j\in\{1,\dots,t-1\}$ consider the progression that consists of the first term of $p_j^{i_1}$ in block B_{i_1}, the second term of $p_j^{i_2}$ in block B_{i_2}, up to the $(k-1)$-st term, which is the $(k-1)$-st term of $p_j^{i_{k-1}}$ in $B_{i_{k-1}}$. The set of these positions forms a progression aiming at a. This way we obtain for any $j \in \{1,\dots,t-1\}$ a progression that aims at a. In total we obtain $t-1$ progressions aiming at a, in addition to the progression $\{a_{B_{i_1}},\dots,a_{B_{i_{k-1}}}\}$ which also aims at a. All t progressions are monochromatic in a different color, since the progressions p_j^i aim at a position in $\{a_{B_{i_1}},\dots,a_{B_{i_{k-1}}}\}$. This shows our claim, and by the definition of $B(k,c,t)$ we conclude the first inequality.

To prove Inequality 3.2 we note that since $B(k,c,c) = w(k;c)$, shown prior to the theorem, any coloring of the set $\{1,\dots,B(k,c,c)\}$ contains a monochromatic arithmetic progression of length k. It suffices therefore to see that if an arithmetic progression $\{a, a+d,\dots,a+(k-1)\cdot d\}$ of length k is contained within the first $n = B(k,c,c)$ positive integers, then its extension to the right, i.e., the progression $\{a,a+d,\dots,a+(k-1)\cdot d, a+k\cdot d\}$, is contained in the first $\frac{k}{k-1}(n-1)+1$ positive integers: The gap of a progression of length k in $\{1,\dots,n\}$ is at most $\frac{n-1}{k-1}$. Thus the last position of the extended progression is at most $\frac{n-1}{k-1}+n = \frac{k}{k-1}(n-1)+1$.

Finally Equations 3.3 and 3.4 are trivial and we have argued Equation 3.5 right before the theorem. □

One may slightly improve the bound of van der Waerden recursion by easy modifications:

- The blocks into which the set S is divided do not have to be disjoint.
- We do not require that the blocks $B_{i_1},\dots,B_{i_{k-1}}$ are colored exactly in the same way. Rather, the only requirement is that the progressions p_j we employ for the proof are at the same position and in the same color. With at most $c^t \cdot B(k,c,t)^t$ ways we can ensure that $t-1$ progressions aim at the same position and specify the involved colors, since such a situation can be described by four properties: By pointing out the position aimed at, the gaps of the involved progressions, the colors of the progressions and the color of the position aimed at, the situation is sufficiently specified.

With these modifications inequality 3.1 of the recursion translates into

$$B(k,c,t) \quad \leq \quad B\left(k, c^t \cdot B(k,c,t-1)^t, 1\right) + B(k,c,t-1) - 1.$$

We can modify this inequality even further: The gap of the progression formed during step j of this procedure can be at most $B(k,r,j)$. Therefore we get:

$$B(k,c,t) \quad \leq \quad B\left(k, c^t \cdot B(k,c,t-1) \cdot \prod_{j=1}^{t-1} B(k,c,j), 1\right) + B(k,c,t-1) - 1.$$

Still, being a double induction, these recursive bounds yield an upper bound of ackermaniac growth. Following [51] we say a function grows ackermaniac if it asymptotically grows as fast as the Ackermann function. We do not go into further detail as better upper bounds for the van der Waerden numbers are known. We present them next.

3.2 Upper bounds for van der Waerden numbers

The ackermaniac upper bound resisted improvement attempts for over 60 years, until in 1988 Shelah [118] showed that the van der Waerden numbers are primitive recursive. The proof is elementary combinatorial and insightful to read. A very illustrative and accessible exposition can be found in [51]. More specifically the proof bounds the numbers $w(k;c)$ by a function that lies in the fifth level of the Grzegorczyk hierarchy.

At present the best known upper bounds are given by Gowers [49]. His analytical proof of Szemerédi's theorem shows that

$$w(k;c) \leq 2^{2^{c^{2^{2^{(k+9)}}}}}.$$

This bound on the van der Waerden numbers also gives us, using the brute force algorithm, an upper bound on the running times necessary to compute the numbers exactly. The brute force algorithm for $n \in \mathbb{N}$ enumerates all proper colorings of $\{1,\ldots,n\}$. For each n there are c^n colorings in total. The smallest n for which there is no coloring that avoids monochromatic progressions is the sought van der Waerden number.

3.3 Lower bounds for van der Waerden numbers

We now consider lower bounds for the van der Waerden numbers. A weak lower bound can readily be obtained with the first moment probabilistic method [2]: The basic idea is to randomly color the integers in some interval $\{1,\ldots,n\}$ and then show that the expected number of monochromatic arithmetic progressions is less than 1. In this case one may conclude that there exists a result of the random coloring experiment, i.e., a coloring which contains no arithmetic progression. Alternatively the same lower bounds can be obtained with the incompressibility method:

Theorem 21 (lower bound for van der Waerden numbers [Erdős, Rado [37] (1952)]). *For the van der Waerden number $w(k;c)$, (i.e., for k-term monochromatic arithmetic progressions of integers colored with c colors), the following inequality holds:*

$$w(k;c) > \sqrt{k} \cdot c^{\frac{k}{2}-1}.$$

Proof. We formulate the proof in the setting of strings: In this setting a coloring $\chi\colon \{1,\ldots,n\} \to \{1,\ldots,c\}$ corresponds to the string $\chi(1),\ldots,\chi(c)$ that consists of characters in the color set $\{1,\ldots,c\}$.

We use the basic fact, that strings cannot be compressed: There is no injective map from the set of strings of length n to the set of strings of length n' for any $n' < n$. We define an injection from strings of length $n = w(k;c)$ to strings of some length n'. We therefore conclude that $n' \geq n$ and use this fact to bound the van der Waerden number $w(k;c)$.

Let s be a string of length $n = w(k;c)$. By definition, s contains a k-term monochromatic arithmetic progression. (In the setting of strings, a monochromatic arithmetic progression is a set of positions in the string that forms an arithmetic progression which only consists of one character.) Number all arithmetic progressions of length k, that occur in a string of length n. There are at most n^2/k such progressions. Using its number encode one monochromatic progression in s (which must exist), by a string e with $\lceil \log_c(n^2/k) \rceil$ characters. Delete it from the string and attach the encoding plus the character t of the progression, using 1 additional character, to the front of the string. Summarizing, we obtain a new string $te\hat{s}$, where $\hat{s}$ is obtained from s by deleting the characters that are contained in the monochromatic arithmetic progression.

Using the same method, we map all strings of length n to a new string of length n' for some fixed integer n'. The obtained mapping is injective, since the operation can be reversed. Therefore, the resulting strings are no shorter than the strings we start with. We get:

$$n' = 1 + \lceil \log_c(n^2/k) \rceil + n - k \geq n,$$

solving for n yields:

$$w(k;c) = n > \sqrt{k} \cdot c^{\frac{k}{2}-1}.$$

□

Instead of mapping all strings s, we may use a string that is incompressible, in the sense of Kolmogorov complexity. We do not go into detail, see Li's and Vitányi's introduction to Kolmogorov complexity [83] for a broad treatment of the theory, including the incompressibility method.

3.3.1 Lovász' Local Lemma in the context of van der Waerden numbers

A better lower bound than the one just presented may be obtained via Lovász' Local Lemma. In this section $e \approx 2.71828$ denotes the Euler constant.

Theorem 22 (Lovász' Local Lemma (symmetric version) [Erdős, Lovász [36] (1975)]). *Let $A_1, \ldots, A_n$ be a series of events in a probability space, such that for each $1 \leq i \leq n$ the event A_i is mutually independent of all but $d \in \mathbb{N}$ of these events, and such that the probability of each A_i is at most $p \in [0,1]$. If $e \cdot d \cdot p \leq 1$, then there is a positive probability that none of the events A_i occur.*

An application of Lovász' Local Lemma improves on the previously stated lower bound for van der Waerden numbers. We obtain:

Theorem 23 (improved lower bound for van der Waerden numbers [51]). *For the van der Waerden number $w(k;c)$, (i.e., for k-term monochromatic arithmetic progressions of integers colored with c colors), the following inequality holds:*

$$w(k;c) > \frac{c^k}{eck}(1+o(1)).$$

In [117] we show how the same bound (up to a factor of 2) can be obtained by a repeated application of the encoding step used in the proof of Theorem 21.

Szabó [123] uses a variant of Lovász' Local Lemma and non-trivially exploits the fact that most intersecting progressions only meet in one common point. He shows that for any $\varepsilon > 0$ there exists a $k_0(\varepsilon)$, such that for all $k \geq k_0$

$$w(k;2) \geq \frac{2^k}{k^\varepsilon}.$$

Berlekamp [12] uses finite fields to show that for any prime p

$$w(p+1;2) \geq p2^p.$$

Brown, Landman and Robertson [20] show further asymptotic bounds in case some parameters are fixed. Moser [105] recently proves a constructive version of Lovász' Local Lemma, with which one can obtain proper colorings (of essentially the same size as given by the bound). These colorings may then serve as deterministic certificates for lower bounds. Still, algorithms specificly designed to supply lower bounds for van der Waerden numbers yield stronger lower bounds than general techniques.

Due to their monotonicity, upper and lower bounds for van der Waerden numbers can be obtained from the bounds mentioned in this and the previous section. We now consider exact values of mixed van der Waerden numbers.

3.4 Known mixed van der Waerden numbers

Few exact values of mixed van der Waerden numbers are known. Recall that by Definition 32 the integer $w(k_1,\ldots,k_c;c)$ is the mixed van der Waerden number, for which arithmetic progressions may not have length k_t if the progressions are monochromatic in color t.

For $c = 1$ we conclude from the definition that $w(k_1;1) = k_1 + 1$. For an arbitrary number of colors c and any choice of lengths $k_1,\ldots,k_c$, we conclude directly from the definition that $w(k_1,\ldots,k_c,1;c+1) = w(k_1,\ldots,k_c;c)$. Furthermore under any permutation π of the colors $\{1,\ldots,c\}$, the van der Waerden number does not change, i.e., $w(k_1,\ldots,k_c;c) = w(k_{\pi(1)},\ldots,k_{\pi(c)};c)$. We therefore always sort the lengths such that $k_1 \leq \ldots \leq k_c$ and restrict the entries to be greater than 1.

For van der Waerden numbers of the form $w(2,\ldots,2,k_c;c)$, there exists an explicit formula for cases where k is large enough (where large enough is a function depending on c), as shown by Culver, Landman and Robertson [80].

$w(k_1, k_2; 2)$															
$k_1\downarrow k_2\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	6	7	10	11	14	15	18	19	22	23	26	27	30	31	34
3	9	18	22	32	46	58	77	97	114	135	160	186	218	238	279
4	18	35	55	73	109	146									
5	22	55	178	206											
6	32	73	206	*1132*											

$w(2, k_2, k_3; 3)$															
$k_2\downarrow k_3\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	7	11	15	16	21	22	25	29	33	34	39	40	43	47	51
3	14	21	32	40	55	72	90	108	129	150	171	**202**			
4	21	40	71	82	119										
5	32	71	180												
$w(3, k_2, k_3; 3)$															
$k_2\downarrow k_3\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	27	51	80												
4	51	89													

$w(2, 2, k_3, k_4; 4)$															
$k_3\downarrow k_4\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
2	8	12	20	21	28	29	32	34	44	45	52	53	56	58	68
3	17	25	43	48	65	83	99	119	**141**						
4	25	53	75	93											
$w(2, 3, k_3, k_4; 4)$															
$k_3\downarrow k_4\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	40	60	86												
$w(3, 3, k_3, k_4; 4)$															
$k_3\downarrow k_4\rightarrow$	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
3	76														

Figure 3.2: For $c \leq 4$ the figure shows the known mixed van der Waerden numbers $w(k_1, \ldots, k_c; c)$ for $2 \leq k_1 < \ldots < k_c < 17$ and $k_c > 2$. With the exception of $w(6, 6; 2) = 1132$, all numbers have been computed with the wildcards algorithm explained in Section 3.8. The two numbers shown in bold were previously not known.

In particular we get the explicit formula

$$w(2, k_2; 2) = \begin{cases} 2k_2 - 1 & \text{if } k_2 \text{ is even,} \\ 2k_2 & \text{if } k_2 \text{ is odd.} \end{cases}$$

Figure 3.2 summarizes the known mixed van der Waerden numbers. These numbers were computed with various methods over time, with increasingly more powerful approaches and computing machinery, by Chvátal [26], Brown [19], Stevens and Shantaram [121], Beeler and O'Neil [11], Beeler [10], Landman, Robertson and Culver [80], Kouril [73] and Kouril and Paul [74]. Recently Ahmed [1] has posted mixed van der Waerden numbers, which are also included in the figure. With the exception of $w(6, 6; 2) = 1132$ all numbers have been verified with the wildcards algorithm (see Section 3.8).

3.5 Detecting monochromatic arithmetic progressions

As mentioned in the beginning of this chapter, the structure of the underlying object, when computing van der Waerden numbers, i.e., that of arithmetic progressions, is simple. The number of arithmetic progressions contained in $\{1, \ldots, n\}$ is at most cubic in n, as each progression is well defined by its first two terms and its length.

Erikson [38] uses a dynamic programming approach to obtain an algorithm that computes the longest arithmetic progression within a set of n integers in $\mathcal{O}(n^2)$ time. There is a matching lower bound of $\Omega(n^2)$ in the 3-linear decision model for any algorithm that computes the longest arithmetic progression. Erikson's algorithm can also be used to determine whether a given coloring of the set $\{1, \ldots, n\}$ contains a monochromatic arithmetic progression of a certain color and length. We describe such an adaption next.

The algorithm proceeds as follows: Given a certain color $t \in \{1, \ldots, c\}$ it computes iteratively for every pair $i, j \in \{1, \ldots, n\}$ with $i < j$ the value $L(i, j)$, the length of the longest monochromatic arithmetic progression in color t with first term i and second term j. (If position i is not colored in t then $L(i, j) = 0$. If position i is colored in t, but position j is not colored different in t then $L(i, j) = 1$). The key observation is the following: Assume that j, d are positive integers such that $j - d, j, j + d \in \{1, \ldots, n\}$. Then $L(j - d, j) = L(j, j + d) + 1$, if all three integers $j - d, j, j + d$ are colored with t. Algorithm 7, very similar in fashion to Erikson's algorithm, runs a quadratic loop over position j and gap $d = j - i$ to compute these values $L(i, j) = L(j - d, j)$.

It is also possible to modify the algorithm, so that it computes the longest monochromatic progression for all colors simultaneously.

Erikson also explains how to improve this algorithm, if the length ℓ of the longest arithmetic progression is relatively large compared to n, the number of integers that are colored. Since in all our applications the van der Waerden number $n = w$ is large in comparison to the lengths of the progressions $\ell = k_t$, this is not of use to us.

Maintaining the values $L(i, j)$ avoids a repeated recomputation of arithmetic progressions. In particular the culprit algorithm, which we consider next, can benefit

Algorithm 7 Longest progression of a certain color

```
Input: A coloring $\chi$ of $\{1,\dots,n\}$ and a color $t$
Output: The length $\ell$ of the longest arithmetic progression in color $t$.
$\ell \leftarrow 0$
for $j = n$ down to 1 do
  if $\chi(j) = t$ then
    $\ell \leftarrow \max\{\ell, 1\}$
  end if
  for $d = 1$ to $j - 1$ do
    if $\chi(j-d) \neq t$ then
      $L(j-d, j) \leftarrow 0$
    else if $\chi(j) \neq t$ then
      $L(j-d, j) \leftarrow 1$
    else if $j + d \leq n$ and $\chi(j+d) = t$ then
      $L(j-d, j) \leftarrow L(j, j+d) + 1$ // $L(j, j+d)$ has previously been computed
    else                                   // $j + d > n$ or $\chi(j+d) \neq t$
      $L(j-d, j) \leftarrow 2$
    end if
    $\ell \leftarrow \max\{\ell, L(j-d, j)\}$
  end for
end for
return $\ell$
```

from this maintaining the values.

3.6 The culprit algorithm

We now turn to algorithms that compute mixed van der Waerden numbers. Suppose we want to compute the van der Waerden number $w(k_1, \dots, k_c; c)$. A simple brute force method to determine a van der Waerden number w has to try c^w colorings of $\{1, \dots, w\}$. This rapidly becomes infeasible. The technique of coloring initial segments and discarding them, whenever they contain a monochromatic arithmetic progression that is too long, reduces the number of colorings that have to be considered, but still yields infeasible running times for all but a few van der Waerden numbers.

The culprit algorithm, as first appeared in [11] and described in [81], improves over the brute force algorithm by making use of the following observation: Assume we know a lower bound w_{lb} for the van der Waerden number we are about to compute. We also assume that an initial interval $\{1, \dots, i\}$ with $i \leq w_{\text{lb}}$ is already colored. If there is a position i' with $i < i' \leq w_{\text{lb}}$ such that for every color $t \in \{1, \dots, c\}$ a monochromatic progression of length $k_t - 1$ aims at that position i', then our current initial coloring cannot be extended past position $i' - 1$. (Recall that, as defined in Subsection 3.1, a progression $\{a, a+d, \ldots, a+(k-1)\cdot d\}$ aims at position $a+k\cdot d$). Position i' is called the culprit, as it does not allow the coloring to continue. Algorithm 8, using this observation, recursively enumerates all proper colorings. It continuously updates its lower bound whenever a proper coloring exceeding the current bound has been found. For illustrative purposes the algorithm is specialized to the case $k = k_1 = k_2 = \ldots = k_c$. It is easily adapted to the general case. Besides the number of colors c and the length k as natural inputs, it takes as input a lower bound w_{lb} and an initial interval $\{1, \ldots, i\}$ that has been properly colored with a coloring χ. To compute $w(k;c) = w(k, k, \ldots, k; c)$, the algorithm is called with parameters $(0, \chi, 0, c, k)$, where $\chi\colon \{\} \to \{1, \ldots, c\}$ is the coloring of the empty set.

Algorithm 8 Culprit algorithm [81]

Input: $(i, \chi, w_{\mathrm{lb}}, c, k)$: A proper coloring χ of $\{1, \ldots, i\}$, a lower bound w_{lb}, the number of colors c and a desired length k.

Output: w_{lb} is the largest integer for which there exists an extension of χ to $\{1, \ldots, w_{\mathrm{lb}}-1\}$ avoiding monochromatic progressions of length k.

```
1: for $t = 1$ to $c$ do
2:    $\chi(i+1) \leftarrow t$
3:    if there is no monochromatic progression of length $k$ in $\{1, \ldots, i+1\}$ and there
      is no culprit $i' \in \{i+2, \ldots, w_{\mathrm{lb}}\}$ then    // i.e, a position aimed at by $c$
      differently colored monochromatic arithmetic progression of length $k-1$
4:       $w_{\mathrm{lb}} \leftarrow \max\{w_{\mathrm{lb}}, i+1\}$
5:       $w_{\mathrm{lb}} \leftarrow$ Culprit algorithm$(i+1, \chi, w_{\mathrm{lb}}, c, k)$
6:    end if
7: end for
8: return $w_{\mathrm{lb}}$
```

3.7 Kouril's and Paul's SAT technique

In [74] Kouril and Paul use SAT techniques, previously developed in Kouril's thesis [73], to compute the van der Waerden number $w(6,6;2)$. (Earlier [33] also describes applicability of SAT solvers to determine mixed van der Waerden numbers.) The main technique is to encode the requirement that monochromatic arithmetic progressions must be avoided into a Boolean expression in conjunctive normal form. For every position $i \in \{1, \ldots, n\}$ and every color $t \in \{1, \ldots, c\}$, a variable $x_{i,t}$ with values in $\{\mathbf{true}, \mathbf{false}\}$ determines whether i is colored with color t. Clauses are introduced to guarantee that for every i exactly one $x_{i,t}$ is true. Furthermore, for every progression (of the length in question) a clause guarantees that the positions cannot all be colored with the same color. In case only two colors are available, this clause requires that every color must appear in one of the variables of the clause. Given an initial coloring of the variables, one may invoke a SAT solver to determine whether there is an extension of a certain length. The SAT solver used for the computation of $w(6,6;2)$

is restricted to inferences and contradictions. In the terminology of SAT problems the culprit algorithm from the previous section does exactly this: It checks whether there are inferences that contradict each other.

The second major ingredient in the method, referred to as preprocessing, is the determination of a set of unavoidable patterns, of which at least one must occur within any feasible coloring of an interval of integers of sufficient length.

3.8 The wildcards algorithm for mixed van der Waerden numbers

We now present an algorithm that computes, for any number of colors $c \in \{1, 2, \ldots\}$ and arbitrary lengths $k_1, \ldots, k_c \in \{1, 2, \ldots\}$, the mixed van der Waerden number $w(k_1, \ldots, k_c; c)$. Throughout this section we fix the parameters c and $k_1, \ldots, k_c$. We call the algorithm wildcards algorithm because of the main idea that we exploit: For many non extremal colorings, (i.e., colorings of a set of consecutive integers shorter than the maximal possible length), there are many positions that may be recolored with a different color, without introducing a monochromatic arithmetic progression. We use wildcards as placeholders to indicate that the color of these positions is not pinpointed by the coloring of the other positions. (In the words of satisfiability, for two colors this is a form of delayed evaluation.) To handle these placeholders, we define varicolorings:

Definition 34 (varicoloring of integers). A map $\lambda\colon \{1, \ldots, n\} \to \mathcal{P}(\{1, \ldots, c\}) \setminus \{\}$, i.e, a map into the power set of the colors, is said to be a *varicoloring* of the interval $\{1, \ldots, n\}$ with c colors.

For disambiguation we refer to the elements of $\{1, \ldots, c\}$ as ordinary colors and refer to maps into $\{1, \ldots, c\}$ as ordinary colorings. We usually denote ordinary colors by t and varicolors by T. When presenting examples we use two colors, call them red and blue, and refer to the varicolor $\{\text{red}, \text{blue}\}$ as magenta. In contrast to the concept of varicolorings, a multicoloring is a coloring (either a varicoloring or an ordinary coloring) for which $c > 2$, i.e., a coloring that is allowed to use more than two ordinary colors.

Definition 35 (coarser, finer, specification). Given two varicolorings

$$\lambda, \lambda'\colon \{1, \ldots, n\} \to \mathcal{P}(\{1, \ldots, c\}) \setminus \{\}$$

we say that λ is *coarser* than λ', (and λ' is *finer* than λ') if

$$\forall i \in \{1, \ldots, n\}\colon\ \lambda'(i) \subseteq \lambda(i).$$

In this case we also say that λ *specifies* to λ'.

Further, we say that a set of varicolorings $\mathcal{L}$ *covers* another set of varicolorings $\mathcal{L}'$, if $\mathcal{L}$ is an upper bound for $\mathcal{L}'$, i.e., every element in $\mathcal{L}'$ is a specification of some element in $\mathcal{L}$.

A varicoloring λ is said to be *proper* with respect to the color lengths $k_1, \dots, k_c$ if no specification of λ to an ordinary coloring is improper, i.e., if it does not specify to an ordinary coloring that contains a monochromatic arithmetic progression of length k_t in some color $\{t\}$, with $t \in \{1, \dots, c\}$. (Here we implicitly identify the ordinary colorings with the varicolorings that use only varicolors that are sets of size 1.) The set of varicolorings forms a partial order in which the proper varicolorings form a suborder. The minimal elements are the ordinary colorings.

A varicoloring thus models a set of ordinary colorings. The proper varicolorings are the varicolorings of interest to us, since they simultaneously model proper ordinary colorings. By handling several ordinary progressions simultaneously, we reduce the amount of work carried out by an algorithm that computes van der Waerden numbers.

The notion of varicolorings is applicable to any category of colored objects. Since we do not consider isomorphisms between colorings of integers (which could be reversal or permutation of colors), we do not require this categorical view. We postpone this view to Chapter 4.

With the given terminology we may describe the wildcards algorithm: The algorithm iteratively for n ranging from 1 to w, the sought van der Waerden number, constructs an antichain $\mathcal{L}_n$ (i.e., a set of pairwise incomparable elements) in the partially ordered set of proper varicolorings of $\{1, \dots, n\}$. The constructed list $\mathcal{L}_n$ covers all ordinary proper colorings of $\{1, \dots, n\}$. In order to keep the size of this antichain small, we only include maximal proper varicolorings, i.e, varicolorings that are maximal among the set of proper varicolorings, into the list. (Taking maximal elements is only a heuristic strategy. It is not optimal since the antichain of minimal size that covers all proper ordinary colorings may contain colorings that are not maximal.)

When we extend a proper varicoloring λ of $\{1, \dots, n\}$ to a proper varicoloring λ' of $\{1, \dots, n+1\}$, it suffices to check that no specification of λ' forms monochromatic arithmetic progressions *that contains position* $n + 1$: If λ' were to specify to an ordinary coloring that contains a monochromatic arithmetic progression within $\{1, \dots, n\}$, then λ would also specify to an improper ordinary coloring.

Given a varicoloring $\lambda \colon \{1 \dots, n\} \to \mathcal{P}(\{1, \dots, c\}) \setminus \{\}$, we define for any position $i \in \{1, \dots, n\}$ and varicolor $T \subseteq \{1, \dots, c\}$ the *recoloring of* λ *of position* i *with color* T as the varicolor $\lambda_{i \to T} \colon \{1 \dots, n\} \to \mathcal{P}(\{1, \dots, c\}) \setminus \{\}$ given by

$$\lambda_{i\to T}(j) := \begin{cases} \lambda(j) & \text{if } j \neq i, \\ T & \text{if } j = i. \end{cases}$$

We define two properties a potential color at a certain position in a given varicoloring may have. They are depicted in Figure 3.3:

Definition 36 (prohibited, innocuous). Let $\lambda \colon \{1, \dots, n\} \to \{1, \dots, c\}$ be a varicoloring. Let $t \in \{1, \dots, c\}$ be an ordinary color, $i \in \{1, \dots, n\}$ a position and $\lambda_{i \to \{t\}}$ be the recoloring of position i with color $\{t\}$.

- We say the ordinary color t is *prohibited* at position i if $\lambda_{i\to\{t\}}$ contains a monochromatic arithmetic progression of color $\{t\}$ that contains position i.

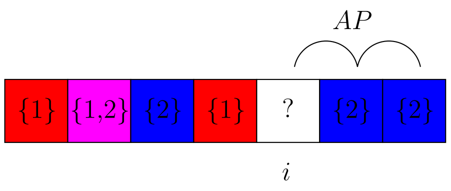

Figure 3.3: For two colors, (i.e., $c = 2$), and lengths 3, (i.e., $k_1 = k_2 = 3$), the figure depicts a varicoloring for which blue $= 2$ is prohibited at position i: If position i is recolored with blue then progression AP is monochromatic in blue. On the other hand, red $= 1$ is innocuous for position i.

- We say that color t is *innocuous* at position i, if $\lambda_{i\to\{t\}}$ does not specify to any coloring which contains a monochromatic arithmetic progression of color $\{t\}$ that contains position i.

Given a list $\mathcal{L}_n$ of proper varicolorings of $\{1, \dots, n\}$ which covers all ordinary proper colorings of $\{1, \dots, n\}$, we now explain how to construct a list $\mathcal{L}_{n+1}$ of proper varicolorings of $\{1, \dots, n+1\}$ which covers all ordinary proper colorings of $\{1, \dots, n+1\}$.

For every varicoloring $\lambda \in \mathcal{L}_n$, we construct its extension: We define $\lambda' := \lambda_{n+1\to C}$ as the extension of λ to the set $\{1, \dots, n+1\}$ defined by

$$\lambda_{n+1\to C}(i) := \begin{cases} C & \text{if } i = n+1 \\ \lambda(i) & \text{if } i < n+1, \end{cases}$$

where $C = \{1, \dots, c\}$ is the set of colors. If λ' is proper, then we insert it into the list $\mathcal{L}_{n+1}$. Otherwise, i.e. if λ' is improper, we find a set of specifications of λ' which covers all proper ordinary colorings that are covered by λ'. These specifications are then added to the list $\mathcal{L}_{n+1}$. We find the specifications by performing a backtracking: If λ' is improper, we first remove prohibited colors: Suppose position i is colored with color T, which contains a color $t \in T$ that is prohibited for position i. We recolor position i by removing t and obtain the new varicoloring $\lambda'_{i\to T\setminus\{t\}}$. After we repeatedly remove prohibited colors, we suppose there is no prohibited color in λ' (for any position). If λ' is now proper we add it to the list $\mathcal{L}_{n+1}$. Otherwise there are two possibilities. Either λ' does not cover any proper ordinary coloring, in which case we discard λ', or there is a position $i \in \{1, \dots, n+1\}$ colored with varicolor T say, for which the following holds: The set T contains at least two ordinary colors and not all ordinary colors contained in T are innocuous at position i. If such a position i exists, we branch by splitting T into one part that contains the innocuous colors, and several parts that each consist of one color that is not innocuous: Let $T' \subset T$ be the set of innocuous colors in T. We construct the set $\Lambda :=$

$$\{\lambda'_{i\to T'}\} \cup \{\lambda'_{i\to\{t\}} \mid t \in T \setminus T'\}.$$

The varicolorings in the set Λ cover all ordinary proper colorings that are covered by λ'. We add all proper varicolorings in Λ into the list $\mathcal{L}_{n+1}$. For all improper varicolorings in Λ we recurse, i.e. we first remove the prohibited colors, then possibly branch and so on.

Since in every recursive step we split a set of colors, this process ends. In the end we obtain a list $\mathcal{L}_{n+1}$ of proper varicolorings of $\{1, \dots, n+1\}$ which covers all ordinary proper colorings of $\{1, \dots, n+1\}$.

Algorithm 9 describes how to perform the branching in detail. It takes as input a list $\mathcal{L}_n$ of proper varicolorings of length i and produces a list $\mathcal{L}_{n+1}$ that covers all proper extensions of the varicolorings in $\mathcal{L}_n$. The algorithm is called with input $\mathcal{L}_0 = \{\lambda_\varepsilon\}$, where λ_ε is the coloring of the empty set.

Algorithm 9 Wildcards algorithm

Input: A set of colors $C = \{1, \dots, c\}$ and a set $\mathcal{L}_n$ of varicolorings of length n.

Output: A set $\mathcal{L}_{n+1}$ of proper varicolorings of length $n+1$, covering all proper ordinary colorings that are extensions of proper ordinary colorings covered by $\mathcal{L}_n$, i.e., if λ_i is a proper ordinary specification of an element in $\mathcal{L}_n$ and λ_{n+1} is an ordinary proper extension of λ_n, then λ_{n+1} is a specification of an element in $\mathcal{L}_{n+1}$.

```
1:  $\mathcal{L}_{n+1} \leftarrow \{\}$
2:  $S \leftarrow \{\}$
3:  for all $\lambda \in \mathcal{L}_n$ do
4:      $S \leftarrow S \cup \{\lambda_{n+1 \to C}\}$
5:  end for
6:  while $S \neq \{\}$ do
7:      pick $\lambda' \in S$
8:      while there is a position $i$ and an ordinary color $t \in T := \lambda(i)$ prohibited at
        position $i$ do                                   // remove prohibited colors
9:          $\lambda' \leftarrow \lambda'_{i \to T \setminus \{t\}}$
10:     end while
11:     if $\lambda'$ is proper then
12:         $\mathcal{L}_{n+1} \leftarrow \mathcal{L}_{n+1} \cup \{\lambda'\}$
13:     else if there is a position $i$ colored in $T := \lambda(i)$ with $|T| \geq 2$ and not all $t \in T$
        are innocuous at position $i$ then                               // branch
14:         $T' \leftarrow \{t \in T \mid t \text{ is innocuous at position } i\}$
15:         $S \leftarrow S \cup \{\lambda_{i \to T'}\}$
16:         for all $t \in T \setminus T'$ do
17:             $S \leftarrow S \cup \{\lambda_{i \to \{t\}}\}$
18:         end for
19:     end if
20: end while
```

For every length n, the set of varicolorings $\mathcal{L}_n$ that we iteratively construct this way has the favorable property that every proper ordinary coloring is finer than exactly one of the varicolorings in $\mathcal{L}_n$. In the terminology of partial orders, it is a strong downwards antichain. Recall that a *strong downwards antichain* in a partially ordered set P is a subset of the elements which pairwise do not have a common lower bound, i.e., a set $X \subseteq P$, such that

$$\forall x, y \in X, x \neq y \colon \nexists z \in P \colon z \leq y \wedge z \leq x.$$

Lemma 6. *For every n the set $\mathcal{L}_n$ of proper varicolorings forms a strong downward antichain.*

Proof. We show the statement by induction on n. The base case is trivial since $\mathcal{L}_0$ only contains λ_ε, the varicoloring of the empty set. For $n > 0$, every varicoloring in $\mathcal{L}_n$ is constructed from a varicoloring in $\mathcal{L}_{n-1}$. If two varicolorings λ_n, λ'_n from $\mathcal{L}_n$ are constructed from two different varicolorings $\lambda_{n-1}, \lambda'_{n-1}$ in $\mathcal{L}_{n-1}$, then the restrictions to $\{1, \ldots, n-1\}$ of λ_n and λ'_n are specifications of λ_{n-1} and λ'_{n-1}, respectively. Since by induction λ_{n-1} and λ'_{n-1} do not have a common lower bound, i.e., an element finer than both of them, the restrictions of λ_n, λ'_n, and therefore the varicolorings λ_n, λ'_n, do not have a common lower bound.

It suffices thus to show that the new varicolorings obtained by extending one specific $\lambda \in \mathcal{L}_{n-1}$ in the step from $n-1$ to n do not have a common lower bound. This is true since whenever we branch at a position i the colors of that position are partitioned into disjoint sets of colors. More formally, in the execution of Algorithm 9, after every iteration of the main while loop, no two varicolorings in the set $S \cup \mathcal{L}_n$ specify to the same ordinary coloring. □

Given a set $\mathcal{L}_n$, it is consequently very easy to compute the number of proper ordinary colorings of the set $\{1, \ldots, n\}$. Indeed, since the elements of $\mathcal{L}_n$ form a strong downward antichain, we simply sum over the number of proper ordinary colorings covered by each individual $\lambda \in \mathcal{L}_n$. To do so for each $\lambda \in \mathcal{L}_n$ we multiply the sizes of the color sets of all positions. In particular $\mathcal{L}_n$ is empty if and only if for w, the corresponding mixed van der Waerden number $w \leq n$ holds.

The algorithm in this form can already quickly compute most known mixed van der Waerden numbers. To enable the algorithm to compute more van der Waerden numbers, we tweak the algorithm and find a fast implementation.

3.8.1 Incorporating culprits in the wildcards algorithm

To improve efficiency we incorporate the culprit technique from Section 3.6 into the wildcards algorithm. As before, the algorithm needs to maintain a lower bound that is continuously updated. If we detect that there is a position $i \leq w_{\mathrm{lb}}$ aimed at by a monochromatic arithmetic progression of length $k-1$, in every ordinary color, we conclude, that the current varicoloring cannot be extended beyond the lower bound. We thus do not consider any extensions of the current varicoloring. (As usual a monochromatic progression must be monochromatic in an ordinary color, i.e., all positions must be equal to the same color set of size one.)

The wildcards algorithm properly colors increasingly large initial intervals of the form $\{1, \ldots, n\}$. Given a lower bound w_{lb}, we may also start to color intermediate intervals $\{n', \ldots, n\}$ with $1 \leq n' \leq n \leq w_{\text{lb}}$, and extend these to larger and smaller integers. The advantage of this is that we may use the culprit argument toward both directions, which further reduces the running time.

Instead of placing the intermediate coloring at some fixed position within the larger colorings, we may vary its position in every instance. Instead of considering all vari-colorings in the list $\mathcal{L}_{n-n'+1}$, we consider a smaller list $\overline{\mathcal{L}}_{n-n'+1}$. Intuitively, we shift the positions by s and color the set $\{n'+s, \ldots, n+s\}$ for varying $s \in \mathbb{Z}$. The technique opens a diversity of new possibilities. For example, as mentioned in Section 3.7, we can make use of unavoidable patterns. This brings us to what we call preprocessing.

3.9 Preprocessing techniques

By the term preprocessing, we capture the entirety of methods that decrease the search space by "combinatorial reasoning." Rather than making this notion more precise, we give a flavor of such combinatorial arguments:

- Starting with an unavoidable subpattern: We start with a certain pattern, and allow partial colorings to be extended to either side.
- Pruning equivalent colorings: If the lengths for two colors k_t and $k_{t'}$ are equal, interchanging colors t and t' does not change anything (e.g, coloring $\{1, 2, 3\}$ with $(\text{red}, \text{blue}, \text{red})$ or $(\text{blue}, \text{red}, \text{blue})$ yields equivalent colorings if $k_{\text{red}} = k_{\text{blue}}$). In particular intererchanging the colors t and t' does not change the extendibility of a coloring. Hence, given two colorings that are equivalent under the interchanging of colors that correspond to equal lengths, we may dispose of one of the colorings.
- Using known van der Waerden numbers: When computing $w(k_1, \ldots, k_{c-1}, k_c; c)$, if we know $w' = w(k_1, \ldots, k_{c-1}, k'_c; c)$ for some $k'_c < k_c$, we also know that any proper coloring must have an arithmetic progression of length k'_c monochromatic in color c in any subinterval of length w'.

Even though these search space reductions are usually performed in advance, hence the name preprocessing, some pruning methods may only be performed during the execution of the main algorithm.

We now describe a specific preprocessing technique that is performed during the execution of the main algorithm. There are three aspects to the technique, which we gradually explain with the help of an example.

Assume we want to enumerate, i.e. explicitly construct, all $(5, 5; 2)$-colorings of a certain length, say 170. (Note that $w(5, 5; 2) = 178$ and recall that a $(5, 5; 2)$-coloring is a 2-coloring that avoids monochromatic arithmetic progressions of length 5). We set aside for the moment the fact that our algorithm computes varicolorings and only

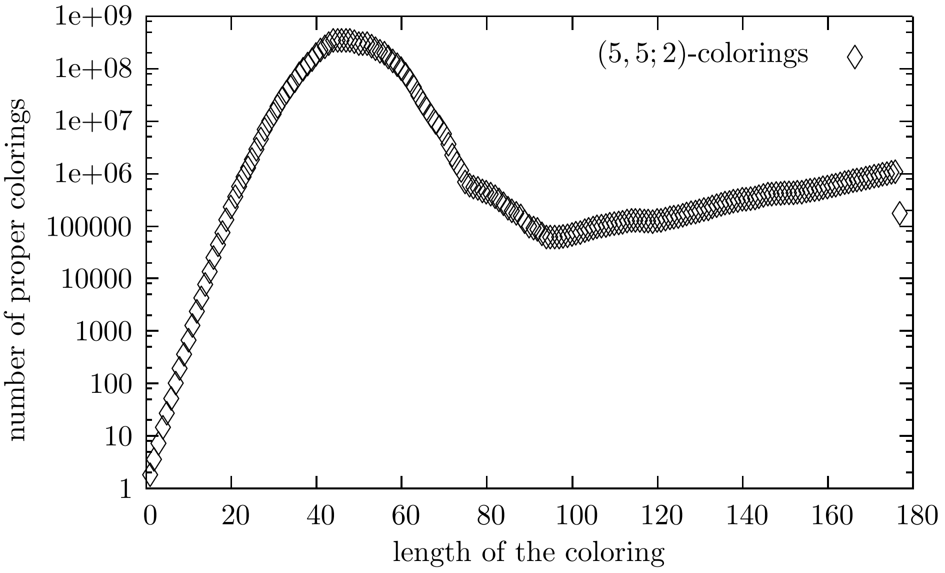

Figure 3.4: The figure depicts the number of proper ordinary $(5, 5; 2)$-colorings for lengths ranging from 1 to 177, in a logarithmic scale. We observe that for small lengths the number exponentially increases, hits a peak, after which it exponentially decreases. It then starts increasing again, up to the number of proper colorings of maximal lengths 177, which is an outlier.

consider ordinary colorings. As is customary, we consider the 2-colorings as 0-1-strings, where 0 and 1 represent the two colors used.

The first aspect of the preprocessing is that it produces restricted lists $\overline{\mathcal{L}}_1, \dots, \overline{\mathcal{L}}_{163}$ of varicolorings, and then, given $\overline{\mathcal{L}}_{163}$, computes a complete list $\overline{\mathcal{L}}_{170}$ of varicolorings, covering all proper ordinary colorings of length 170:

We know that in the first 5 positions color 0 must occur. Within the next 4, following the first 0, the color 1 must occur. Thus the pattern 01 must occur within the first 9 positions.

Instead of starting with just any coloring at position 1, we might therefore insist that our coloring starts with 01, if we allow a shift of up to 9-2 = 7: To explain precisely what we mean by this, we first need the following definition.

Definition 37 (shifted coloring). For a varicoloring $\lambda\colon \{1, \dots, n\} \to \mathcal{P}(\{1, \dots, c\}) \backslash \{\}$ and integers $s, n' \in \mathbb{N}$, with $n' \geq s + n$, we define *the shift of* λ *by* s *within* $\{1, \dots, n'\}$ as the varicoloring $\lambda_{s,n'}\colon \{1, \dots, n'\} \to \mathcal{P}(\{1, \dots, c\})$ given by

$$\lambda_{s,n'}(i) := \begin{cases} \lambda(i - s) & \text{if } i \in \{s + 1, \dots, s + n\}, \\ \{1, \dots, c\} & \text{otherwise.} \end{cases}$$

Informally $\lambda_{s,n'}$ is obtained by shifting λ by s and filling up with $C = \{1, \dots, c\}$.

Going back to our example, we first iteratively for $n \in \{1, \dots, 163\}$ generate the lists $\overline{\mathcal{L}}_n$ that are restricted to $(5, 5; 2)$-colorings that start with 01. (For $\overline{\mathcal{L}}_1$ We only consider the coloring that colors the position 1 with 0.) Each lists $\overline{\mathcal{L}}_n$ covers all proper ordinary colorings that start with 01. Thus we now suppose we have computed a list $\overline{\mathcal{L}}_{163}$ covering all proper ordinary colorings that start with 01 of length $170 - 7 = 163$. From the list we generate the list $\mathcal{L}_{170}$ by computing for all$s \in \{0, \dots, 7\}$ and for all $\lambda \in \overline{\mathcal{L}}_{163}$ all proper specifications of $\lambda_{s,170}$.

To explain the benefit from this indirect computation, we first need to understand how the number of $(5, 5; 2)$-colorings of $\{1 \dots, n\}$ changes with n.

Figure 3.4 shows the number of $(5, 5; 2)$-colorings of lengths 1 to 177 in a logarithmic scale. The characteristics of the function that describes the number of proper colorings (which are of interest to us) are roughly the same for all sets of small parameters $(k_1, \dots, k_c; c)$ (see also Figures 3.5 and 3.7): For small lengths we observe exponential growth. The slope then levels off, until it hits a peak, after which the number of proper colorings significantly decreases. Though the number of proper colorings then may increase again, it never returns to the magnitude attained at the peak.

When we generate $\mathcal{L}_{170}$ with the indirect method described above, for all values n of up to 163 we have reduced the number of colorings generated in our enumeration by a certain fraction, since we only consider colorings that start with 01. Since there are only few colorings of length 163 and beyond, we do not fear the extra work we have from computing the shifted colorings $\lambda_{s,170}$.

As the values around the peak are at least by an order of magnitude larger than any other values, when computing the mixed van der Waerden numbers we have to avoid dealing with most colorings of lengths close to the peak.

The second aspect of our preprocessing technique explains how we can restrict ourselves to subpattern, even if we do not know that they occur within the first few positions of the coloring. To improve the preprocessing technique, we consider (similar to what was used by Kouril and Paul, see Section 3.7) the following: Assume we are guaranteed that the pattern 000 occurs in all $(5, 5; 2)$-colorings of length 170, i.e., that three consecutive positions are colored with 0. Then the pattern 0001 also occurs in all $(5, 5; 2)$-colorings of length 170.

Iteratively for n ranging from 1 to $170/2 = 85$, we generate the lists $\overline{\mathcal{L}}_n$, which consist of colorings which start with 0001 (or a truncated prefix of 0001 for $n < 4$). We then allow a shift of up to 85, i.e., using the previously described technique obtain a list $\overline{\mathcal{L}}_{170}$ of colorings of length 170. Finally we construct the list $\mathcal{L}_{170}$ which consists of all colorings in $\overline{\mathcal{L}}_{170}$ and their reversals, i.e., if the string $\sigma = \sigma_1, \dots, \sigma_{170}$ is in $\overline{\mathcal{L}}_{170}$ then $\mathcal{L}_{170}$ contains σ and its reversal $\sigma^R = \sigma_{170}, \dots, \sigma_1$. With this method even less colorings of lengths below 85 are generated than before. Note that it is necessary to reverse the strings, since we do not know whether the substring 000 is contained in the first part or the second half of the string.

The third aspect improves the preprocessing technique further: Instead of starting with the pattern 000, we generate all colorings (i.e., the colorings are allowed to start with an arbitrary number of zeros). Suppose during the generation of the lists $\overline{\mathcal{L}}_1, \dots, \overline{\mathcal{L}}_{85}$, the current coloring starts with exactly ℓ consecutive zeros. Once we encounter a subpattern that contains more than ℓ consecutive zeros, we declare the coloring as invalid and dispose of it. In other words, we require that the coloring starts with the maximum number of consecutive zeros, and do not consider extensions that produce more consecutive zeros. We again generate a list $\overline{\mathcal{L}}_{85}$, which now contains all proper colorings whose initial segment attains the maximum number of consecutive zeros. From this list we generate $\mathcal{L}_{170}$ as before. Note that it is essential for the correctness argument that all patterns $0 \dots 0 := 0^\ell$, of consecutive zeros, are palindromes. The crucial and trivial fact used here is as follows:

Fact 3. *Any varicoloring of length n that contains a maximum of $\ell > 0$ consecutive zeros contains a subcoloring of length at least $n/2$ that starts with ℓ consecutive zeros, or is a reflexion of such a coloring (i.e., ends with ℓ consecutive zeros).*

In general we use this technique to first compute a list $\overline{\mathcal{L}}_{\lceil w/2 \rceil}$ from which we compute a list $\mathcal{L}_w$. In practice, since we do not know the value w, we replace it by a lower bound w_{lb}. (In the example, the lower bound is 170). Our algorithm first computes the lists $\overline{\mathcal{L}}_1, \dots, \overline{\mathcal{L}}_{\lceil w_{\text{lb}}/2 \rceil}$, then it computes the list $\mathcal{L}_{w_{\text{lb}}}$, and then (without preprocessing technique) all lists $\mathcal{L}_n$ for $n > w_{\text{lb}}$.

When parameters for different ordinary colors $t < t' \in \{1, \dots, c\}$ coincide, i.e., $k_t = k_{t'}$, we may further reduce our search space by disposing of all colorings, which contain more consecutive integers colored with t' than consecutive integers colored with t.

3.9.1 Preprocessing with late peak

For several parameters $(k_1, \dots, k_c; c)$ the "peak" is not significantly smaller than the value $w(k_1, \dots, k_c; c)/2$. In particular, this is the case for the parameter family $(3, k_2; 2)$. Figure 3.5 depicts this phenomenon for $k = 10$. In this case the previous method does not yield the desired search space reduction. But there is a remedy: Intuitively, when we place a coloring of length $\lceil n/2 \rceil$ with some shift s into a coloring of length n, there are $\lfloor n/2 \rfloor$ positions that are unspecified. Therefore, on one of the sides of the placed coloring, at least $\lfloor n/4 \rfloor$ positions must unspecified.

Given a lower bound w_{lb} we proceed by first computing a restricted list $\overline{\mathcal{L}}_{\lceil w_{\text{lb}}/2 \rceil}$ of colorings of length $\lceil w_{\text{lb}}/2 \rceil$ whose initial segment attains the maximum number of consecutive zeros. We then compute a list $\overline{\mathcal{L}}_{\lceil 3w_{\text{lb}}/4 \rceil}$ from the list $\overline{\mathcal{L}}_{\lceil w_{\text{lb}}/2 \rceil}$ by using a shift of *exactly* 0 or *exactly* $\lceil 3w_{\text{lb}}/4 \rceil - \lceil w_{\text{lb}}/2 \rceil \approx w_{\text{lb}/4}$. We obtain the list $\overline{\mathcal{L}'}_{\lceil 3w_{\text{lb}}/4 \rceil}$ from $\overline{\mathcal{L}}_{\lceil 3w_{\text{lb}}/4 \rceil}$ by adding reversals. Finally we compute the list $\mathcal{L}_{w_{\text{lb}}}$ from the list $\overline{\mathcal{L}'}_{\lceil 3w_{\text{lb}}/4 \rceil}$ by using a shift of up to $\lfloor w_{\text{lb}}/4 \rfloor$ and adding reversals.

The following fact, an appropriately modified version of the corresponding Fact 3, guarantees us that the list $\mathcal{L}_{w_{\text{lb}}}$ obtained this way covers all proper colorings.

Fact 4. *Any varicoloring of length n that contains a maximum of $t > 0$ consecutive zeros contains a subcoloring of length at least $n \cdot 3/4$ that*

- *starts with t consecutive zeros,*
- *has t consecutive zeros starting from position $i = \lceil n/4 \rceil$,*

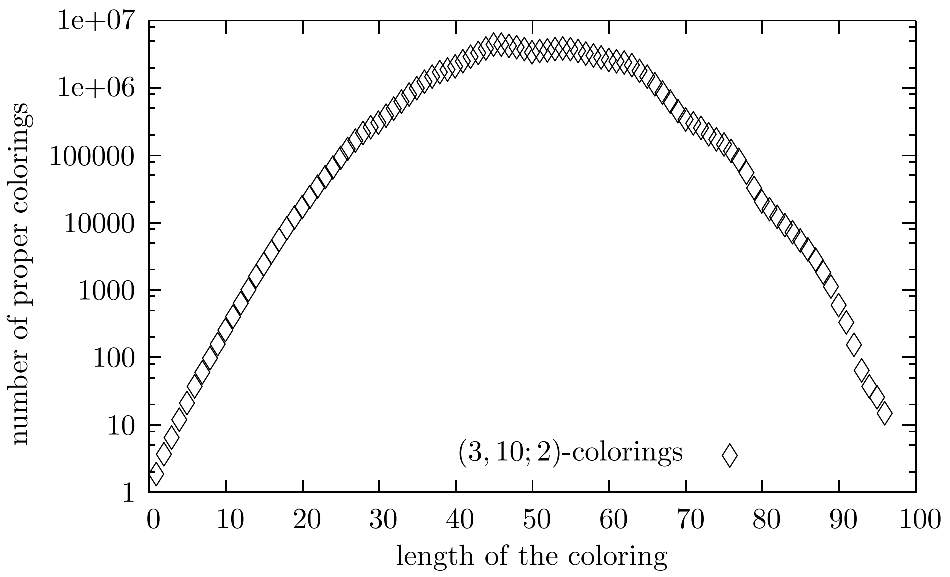

Figure 3.5: The figure depicts the number of proper $(3, 10; 2)$-colorings for lengths ranging from 1 to $w(3, 10; 2) = 97$, in a logarithmic scale. We observe that the peak is attained roughly at $w(3, 10; 2)/2 = 48.5$.

- *or is a reflexion of one of the previous two.*

Using this fact, we may generate restricted lists $\overline{\mathcal{L}}_n$ for n ranging from 1 up to $\lceil 3w_{\mathrm{lb}}/4 \rceil$. (The ratio of lists $\overline{\mathcal{L}}_{w_{\mathrm{lb}}/2}, \ldots, \overline{\mathcal{L}}_{3w_{\mathrm{lb}}/4}$ vs complete lists is roughly a factor of 2 worse than for the lists $\overline{\mathcal{L}}_1, \ldots, \overline{\mathcal{L}}_{w_{\mathrm{lb}}/2-1}$.)

3.9.2 Preprocessing for two colors

We conclude the preprocessing section with a particular preprocessing variant we used for the computation of 2-color van der Waerden numbers: In this variant we consider patterns of the form $0 \ldots 010 \ldots 0 = 0^{\ell}10^{\ell'}$, with $\ell, \ell' \in \mathbb{N}$ and order them lexicographically by two parameters: By the length, (i.e., $\ell + \ell'$), and by the maximum number of consecutive zeros (i.e., $\max\{\ell, \ell'\}$). We thus order the pattern, such that $0^{\ell_1}10^{\ell'_1} \prec 0^{\ell_2}10^{\ell'_2}$ if

$$\ell_1 + \ell'_1 < \ell_2 + \ell'_2 \text{ or } \Big(\ell_1 + \ell'_1 = \ell_2 + \ell'_2 \text{ and } \max\{\ell_1, \ell'_1\} < \max\{\ell_2, \ell'_2\}\Big).$$

For the preprocessing variant, we enumerate all colorings χ for which the initial segment attains the maximum (under "$\prec$") among all subcolorings of χ.

3.10 Implementation details

We briefly mention how varicolors are handled by the wildcards algorithm. To represent the sets of colors, we use integers and encode the subsets by the binary expansion, i.e., if $\{1,\ldots,c\}$ is the set of colors and $\ell = \sum_{i=1}^{c} a_i 2^{i-1} < 2^c$ with $a_i \in \{0,1\}$ is a representation of a varicolor, then ℓ corresponds to the varicolor which is given by $\big\{i \in \{1,\ldots,c\} \mid a_i = 1\big\}$. This way all required operations (such as extension and recoloring of an varicoloring) may be performed with a few integer manipulations.

For illustrative purposes the description of the wildcards algorithm in Section 3.8 describes a breadth first search by generating the lists $\mathcal{L}_n$. However, to maintain a linear space bound, the algorithm has been implemented as a depth first search. The implementation comes with many switches that toggle the use and choice of preprocessing, varicolors, culprits, lowerbounds, double reversing for late peaks and enables distributed computation. A special implementation for two colors avoids overhead coming from the representation as sets. For further details we refer to the code [116].

3.11 Certification

Now that we have designed an algorithm that computes mixed van der Waerden numbers, the question of certification naturally comes up (as it always should). There is an obvious way to certify lower bound claims on the van der Waerden numbers: Extend the output by a coloring that certifies the lower bound, i.e., the algorithm supplies the user with a coloring of the set $\{1\ldots,w-1\}$ that does not have a monochromatic arithmetic progression of respective length. The user may then employ, for example, Erikson's algorithm (i.e., Algorithm 7) to quickly check that the output does not contain monochromatic arithmetic progressions of forbidden lengths. This certification procedure is commonly used in the literature to certify lower bounds.

We define s_1 as the binary string

$$s_1 = 0^4 10^6 10^5 10^3 10^{10} 10^6 10^5 10^8 110^{10} 1010^6 10^7 1010^8 10^{12} 1010^{16} 10^9 110,$$

which is of length 139. The string $s_1 s_1^R$, the concatenation of s_1 with its reversal, serves as a certificate that $w(3,17;2) \geq 279 = 2 \cdot 139 + 1$.

As is the case for extremal colorings for many other parameters, the coloring ss^R is a palindrome. Herwig, Heule, van Lambalgen and van Maaren [61] use the "Cyclic Zipper Method" to exploit regularities in extremal colorings, thereby providing computer verifiable lower bounds on the van der Waerden numbers. Analogously for the van der Waerden number $w(2,3,14;3) = 202$ we set

$$s_2 = 4^{13} 224^3 2424^8 24^4 2424^5 24^8 24244224^9 24^6 224^{11} 24^{12},$$

and obtain the string $s_2 1 s_2^R$ of length 201, consisting of characters 1,2 and 4. In the encoding of the colors, as explained in Section 3.10, color t corresponds to the character 2^{t-1}. The string certifies that $w(2,3,14;3) \geq 202$. In the original version of this thesis, an error in the implementation lead to a miscalculated number and

van der Waerden number	running time in seconds
$w(2,3,11;3)$	2426
$w(2,3,12;3)$	15824
$w(2,3,13;3)$	262057
$w(2,3,14;3)^*$	1229741
$w(2,4,7;3)$	67101
$w(2,5,5;3)$	2602
$w(3,3,5;3)$	19758
$w(2,2,3,9;4)$	2359
$w(2,2,3,10;4)$	27707
$w(2,2,3,11;4)^*$	240534
$w(2,2,4,6;4)$	33507
$w(2,3,3,5;4)$	377600
$w(3,3,3,3;4)$	1218708

Figure 3.6: The figure shows the running times of the wildcards algorithm spent for the computation of various mixed van der Waerden numbers. Running times marked with $*$ are running times that were performed with the revised implementation of the algorithm and on a different machine.

consequently only to a shorter string of length 200. I thank Michal Kouril for pointing out this error to me. He also computed that $w(2,3,14;3) = 202$ and provided me with the string $s_2 1 s_2^R$, proving that the original computation was faulty. The new string has also been computed with a revised implementation of the wildcards algorithm available at [116].

Finally the string

$$s_3 = 8^3(12)848^8448^{10}48^4448^{10}4848^648^728^4448848^94884848^9488418^928828^848^9448^74$$

is a string of length 140, that certifies $w(2,2,3,11;4) \geq 141$. For the varicolor 12, both choices of 4 and 8 yield a proper coloring.

The strings s_1,s_2,s_2' and s_3 were computed with the wildcards algorithm, which moreover showed that the inequalities are exact, i.e., the computation also showed that $w(3,17;2) = 279$, that $w(2,3,14;3) = 202$ and that $w(2,2,3,11;4) = 141$.

Certifying exact values of van der Waerden numbers, however, seems to be intrinsically difficult. One might argue that it involves upper bounds, for the improvement of which we still lack accurate techniques.

3.12 Evaluation and conclusion

The wildcards algorithm has been used to verify all van der Waerden numbers given in Figure 3.2. Two of the values were previously unknown. The van der Waerden

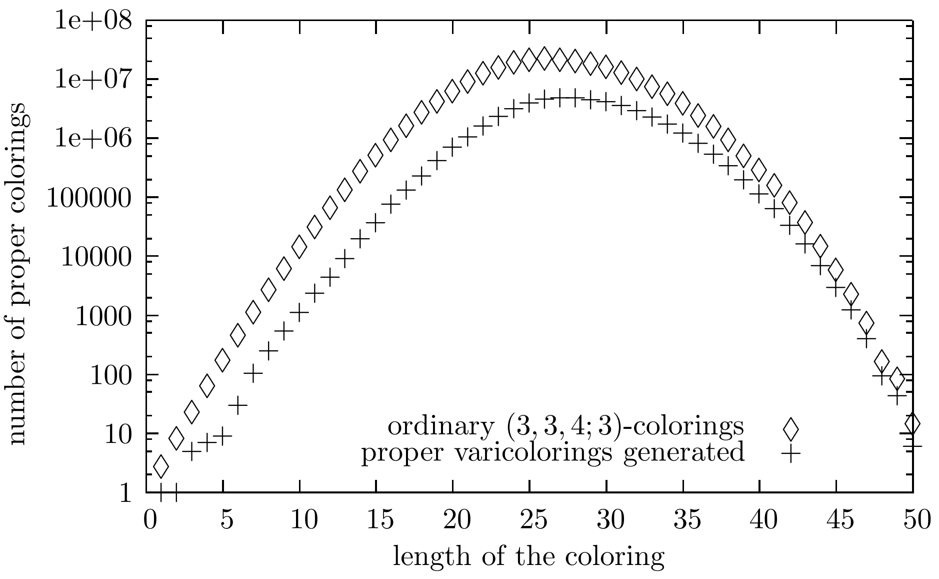

Figure 3.7: Number of $(3, 3, 4; 3)$-colorings in comparison to sizes of the antichains of proper varicolorings produced by the wildcards algorithm without preprocessing for various lengths.

number $w(6, 6; 2)$ has not been computed. Ahmed has posted most of these values at [1], but we do not know what technique was used to compute them.

To see the magnitude of the search space contraction that is achieved by the varicoloring approach, we compare the number of ordinary progressions with the number of varicolors the wildcards algorithm generates. The varicolorings used yield to a search space contraction, as desired. Figure 3.7 shows the number of proper ordinary $(3, 3, 4; 3)$-colorings in comparison to the sizes of the lists $\mathcal{L}_n$ generated by the wildcards algorithm. We observe a reduction of roughly a factor of 10 over the straightforward approach.

It took 552 days of total computation time to compute $w(3, 17; 2)$, which was the longest computation for any of the numbers shown in Figure 3.2. It was divided over 30 clusternodes, each a 2.4 GHz AMD Opteron machine with one 1 GB RAM that runs Linux. Figure 3.6 shows the total computation times for multicolor mixed van der Waerden numbers for various parameters. The values for $w(2, 3, 14; 3)$ and $w(2, 2, 3, 11; 4)$ are the new running times obtained with the corrected code, as explained at the end of Subsection 3.11. These new experiments were performed on 2.4 GHz Intel Xeon E5620 cores. In the meantime these values have also been computed by Kouril.

Unfortunately we do not have any running times available with which to compare. Neither do we have another competitive algorithm available that produces them. We can only stress the fact that our algorithm computed (with the exception of $w(6, 6; 2)$)

the largest mixed van der Waerden numbers known. We remark that for the computation of $w(6, 6; 2)$, various methods of SAT solvers are used. In particular, special hardware (field programmable gate arrays) provided a considerable speedup. Prior to the computation of $w(6, 6; 2)$ in [73], delayed evaluation was used for the computation of mixed van der Waerden numbers. It is related to the varicoloring approach restricted to two colors. The varicoloring approach is in particular well suited for the multicolor case, i.e, when $c > 2$.

Providing upper bounds for the mixed van der Waerden numbers is difficult, both when considering asymptotics, as well as when computing specific numbers. The wildcards algorithm provides a method to compute mixed van der Waerden numbers. To obtain reasonable running times, practical considerations are indispensable, such as preprocessing and algorithm engineering. Though both have been performed with the implementation of the wildcards algorithm, there is still room for improvement. In particular, the use of efficient data structure such as bit-vectors, or hardware such as field programmable gate arrays, may further improve the running times. The computation of the previously unknown mixed van der Waerden numbers $w(2, 3, 14; 3)$ and $w(2, 2, 3, 11; 4)$ serves to show that an implementation of the varicoloring approach can outperform state-of-the-art methods. However, rather than the design of a extremely efficient implementation, the goal pursued in this thesis is the development of general techniques and a framework that may be used for the computation of further Ramsey theory related numbers. These techniques are employed in the next chapter, which explains how the varicoloring technique may be used to compute Ramsey numbers. We will see that computing mixed van der Waerden numbers is theoretically not as involved as computing Ramsey numbers, and we outline several obstacles that we did not encounter in the current chapter.

4 Ramsey numbers

Ramsey's theorem, proven by Frank Ramsey [110] in the year 1930, embodies purely the essential idea, omnipresent in the field of mathematics nowadays called Ramsey theory: Monochromatic substructures are unavoidable when coloring large combinatorial objects with finitely many colors. In this spirit Ramsey's theorem shows that if the edges of a large complete graph are colored with finitely many colors, a monochromatic clique (i.e, a complete subgraph whose edges are all colored equal) must arise. Ramsey numbers quantify the size of the edge colored graph that must be colored in order to guarantee the existence of a monochromatic clique of a specific size (depending on the colors that are used). Though the upper and lower bounds available for Ramsey numbers are closer to the actual values than the bounds for the van der Waerden numbers, already for cliques of size as small 5 the exact computation of the associated 2-color Ramsey number appears very difficult, and has not been performed yet.

In this chapter we use the varicoloring approach, developed in the previous chapter, to outline the wildcards algorithm for Ramsey numbers. The varicoloring approach enables one to simultaneously model different edge colorings of a graph, and thereby achieves a contraction of the search space that has to be traversed, i.e., it reduces the number of colorings that have to be considered. During the design of an efficient implemention, we encounter three major problems. First, the detection of monochromatic substructures, i.e., the cliques which the generated graphs are supposed to avoid is difficult. Second, isomorphism detection is required in order to enumerate colorings without duplicates. A coloring is a duplicate, if it is equivalent to another coloring under permutation of the vertices. Third, to avoid the duplicates the wildcards algorithm maintains a list of colorings, which results in an inadequate space requirement.

Thus though the reduction obtained with the varicoloring approach is promising, the computation of a new Ramsey number cannot be performed, as it first requires efficient algorithms and economical data structures that attack the subproblems.

We proceed in the exposition as follows: We first define and prove the existence of Ramsey numbers (see Section 4.1) and then give upper and lower bounds (see Sections 4.2 and 4.3) as well as the known exact values (see Section 4.4). We describe the drastic sense in which the computational complexity of the problem in unknown (see Section 4.5) and also describe previous algorithms that were used to determine exact values (see Section 4.6). Finally we outline the wildcards algorithm for Ramsey numbers (see Section 4.7) and discuss certification of the output (see Section 4.8).

This chapter, which is held brief, is intended to show the benefits gained from the varicoloring approach, as well as the challenges that arise from using it. It also shows the connection between the two algorithms that were developed in the previous chapters.

4.1 Ramsey numbers

In the previous chapter, our focus lies on colorings of integers and monochromatic progressions within these colorings. Our focus now shifts from integers back to graphs. The basic observation intuitively says that very large graphs cannot simultaneously avoid large cliques and independent sets. Recall that a clique (respectively an independent set) in a graph G is a set of vertices, $K \subset V(G)$ (respectively $I \subset V(G)$), for which $\{\{v, v'\} \mid v, v' \in K\} \subseteq E(G)$ (respectively $\{\{v, v'\} \in E(G) \mid v, v' \in I\} = \{\}$).

Before we make the statement more precise, we first express it in the terminology of colorings: When a large complete graph G is edge colored (see Definition 2) with two colors, it contains a large monochromatic clique, i.e, a subset of vertices $K \subseteq V(G)$ for which all edges in the induced subgraph $G[K]$ are colored equally. The formal statement, generalized to an arbitrary number of colors, is the following:

Theorem 24 (Ramsey's Theorem [Ramsey [110](1930)]). *For any $c \in \mathbb{N}$ and any $k \in \{1, 2, \ldots\}$ there is an integer R such that every edge coloring of K_R (the complete graph on R vertices) with c colors forms a monochromatic clique of size k.*

The theorem is the analogon to van der Waerden's Theorem (see Theorem 19), which deals with colorings of the integers, instead of edge colorings of graphs.

When vertex colorings are considered instead of edge colorings, the equivalent theorem is Dirichlet's pigeonhole principle. The algorithmic problem MAX-CLIQUE(i.e, the task of determining the maximal size of a clique in a graph) is $\mathcal{NP}$-hard. Therefore, detecting the largest monochromatic clique is (presumably) difficult, whereas in the previous chapter, for the detection of the largest monochromatic arithmetic progressions, we have Erikson's algorithm (see Algorithm 7) available, which runs in quadratic time.

As in the previous chapter, we allow the forbidden size of a monochromatic subclique to vary with the color. We are interested in the value of the smallest number R that satisfies the property of Ramsey's theorem:

Definition 38 (Ramsey number). For any $c \in \mathbb{N}$ and any sequence $k_1, \ldots, k_c$ of positive integers define the *Ramsey number* $R = R(k_1, \ldots, k_c; c)$ to be the least integer, for which any edge coloring of K_R (the complete graph on R vertices) contains a monochromatic clique of size k_t in some color $t \in \{1, \ldots, c\}$.

We say that an edge colored complete graph is $(k_1, \ldots, k_c; c)$-*Ramsey*, if it demonstrates a lower bound for the Ramsey number $R(k_1, \ldots, k_c; c)$, i.e., if for every color $t \in \{1, \ldots, c\}$ it does not contain a clique of size k_t monochromatic in color t. We define an *extremal* $(k_1, \ldots, k_c; c)$-*Ramsey graph* as a $(k_1, \ldots, k_c; c)$-Ramsey graph of size $R(k_1, \ldots, k_c; c) - 1$.

We now show the existence of the Ramsey numbers.

4.1.1 Existence of Ramsey numbers

The existence of the Ramsey numbers follows from a decomposition of the Ramsey graphs into Ramsey graphs of smaller parameters:

Theorem 25 (Ramsey recursion). *For $c \in \mathbb{N}$ and any sequence $k_1, \ldots, k_c$ of positive integers, for the Ramsey numbers $R(k_1, \ldots, k_c; c)$ we get*

$$\begin{array}{lcl} R(-;0) & = & 0 \\ R(k_1, \ldots, k_{c-1}, 1; c) & = & R(k_1, \ldots, k_{c-1}; c-1) \\ R(k_1, \ldots, k_c; c) & \leq & 1 + \sum_{t=1}^{c} R(k_1, \ldots, k_{t-1}, k_t - 1, k_{t+1}, \ldots, k_c; c) \end{array}$$

Proof. The first two equations are trivial. To prove the inequality, we consider the complete graph G of size $1 + \sum_{t=1}^{c} R(k_1, \ldots, k_{t-1}, k_t - 1, k_{t+1}, \ldots, k_c; c)$ together with an edge coloring χ. Let $v \in V(G)$ be a vertex. We partition all other vertices according to the color of the edge they form with v. More precisely, we let V_t be the set of vertices $v' \in V(G) \setminus \{v\}$ for which $\chi(\{v, v'\}) = t$. Figure 4.1 illustrates this partition for $c = 2$. By the pigeonhole principle there is a color t, such that

$$|V_t| \geq R(k_1, \ldots, k_{t-1}, k_t - 1, k_{t+1}, \ldots, k_c; c).$$

For this color t, the coloring induced on the subgraph of V_t either contains a monochromatic clique if size $k_{t'}$ for some color $t' \in \{1, \ldots, c\} \setminus \{t\}$ or it contains a monochromatic clique of color t and size $k_t - 1$. In the latter case, the monochromatic clique together with vertex v forms a clique of size k_t monochromatic in color t. Thus any graph of size $1 + \sum_{t=1}^{c} R(k_1, \ldots, k_{t-1}, k_t - 1, k_{t+1}, \ldots, k_c; c)$ is not a $(k_1, \ldots, k_c; c)$-Ramsey graph. □

4.2 Upper bounds for Ramsey numbers

It is known that the Ramsey numbers asymptotically grow exponentially. This follows from the known upper and lower bounds, which we present next.

For two colors, the recursion given in Theorem 25 yields an upper bound for the 2-colored Ramsey numbers given by $R(k_1, k_2; 2) \leq \binom{k_1+k_2}{k_1}$. A proof of this is contained in [51]. Conlon [28] recently published the currently best known bound for the diagonal 2-color Ramsey numbers, i.e., for the case $c = 2$ and $k_1 = k_2$. He shows that there is a constant D, such that for sufficiently large $k \in \mathbb{N}$ we have

$$R(k+1, k+1; 2) \leq k^{-D \frac{\log k}{\log \log k}} \binom{2k}{k}.$$

4.3 Lower bounds for Ramsey numbers

An application of Lovász' Local Lemma (see Theorem 22) provides a lower bound for the diagonal Ramsey numbers as first proven by Spencer in [119]. It shows that

$$R(k, k; 2) > k 2^{k/2} \left(\frac{1}{e\sqrt{2}} + o(1) \right),$$

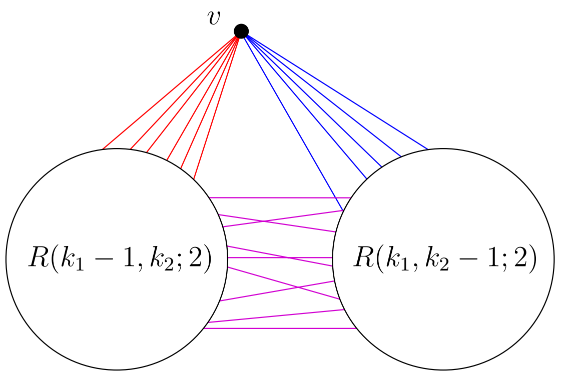

Figure 4.1: An illustration of the proof of the Ramsey recursion (Theorem 25) for the Ramsey number $R(k_1,k_2;2)$. A vertex v partitions all other vertices $v' \neq v$ by the color of the edge $\{v,v'\}$ (shown in red and blue). The size of each partition class is bounded by a Ramsey number for smaller parameters. The horizontal edges that contain a vertex from either partition class are colored either blue or red, (shown in magenta).

where $e \approx 2.71828$ is the Euler constant. See also [51] for a well presented proof.

The lower and upper bounds differ in the base constant when approximated by an exponential function: The central binomial coefficient approximates via Stirling's approximation to $\binom{2k}{k} \approx \frac{1}{\sqrt{\pi n}} 2^{2k}$, thus the constant in the upper bound is $2^2 = 4$, whereas the constant in lower bound is $\sqrt{2}$.

This shows that if there is an approximation for the diagonal Ramsey numbers by an exponential function, the base constant must lie somewhere between $\sqrt{2}$ and 4. If you figure out this exponent do not miss the chance to collect your prize money (see the corresponding problem on diagonal Ramsey numbers at the "Open Problem Garden" [32]).

For the off-diagonal case, Jeong Han Kim [70], and recently Bohman [14] with a direct approach, analyzes the triangle free process to show that the Ramsey numbers $R(3,k;2)$ are of order $\Theta(k^2/\log k)$.

4.4 Known Ramsey numbers

A thorough source for information on values of and bounds on Ramsey numbers is Radziszowski's dynamic survey on small Ramsey numbers [109]. Figure 4.2 summarizes the values of the Ramsey numbers that are exactly known. From the basic identity $R(2,k_2,\ldots,k_c;c) = R(k_2,\ldots,k_c;c-1)$, sequences of k_i that contain an entry equal to 2 can be deduced from shorter sequences. These (infinitely many) values have been omitted in the figure.

$R(k_1, k_2; 2)$							
$k_1 \downarrow, k_2 \rightarrow$	3	4	5	6	7	8	9
3	6	9	14	18	23	28	36
4	9	18	25				

$R(3, k_2, k_3; 3)$							
$k_2 \downarrow, k_3 \rightarrow$	3	4	5	6	7	8	9
3	17						

Figure 4.2: Taken from [109] the figure shows all known values of Ramsey numbers $R(k_1, \ldots, k_c; c)$, with $k_t > 2$ for all $t \in \{1, \ldots, c\}$. The other values with $k_t = 2$ can be obtained from the figure with the identity $R(2, k_2, \ldots, k_c; c) = R(k_2, \ldots, k_c; c-1)$.

4.5 Computational Complexity of Ramsey numbers

The computational complexity of determining Ramsey numbers is (in a drastic sense) unknown. Haanpää shows in his thesis [57] (also see the corresponding technical report [58]) why this is the case. Following Haanpää, we now want to capture the flair of uncertainty we have concerning the computational complexity of our problem.

We first define a more general problem. We use the typical Ramsey arrow notation. For arbitrary graphs G, H_1, H_2 the arrow notation $G \rightarrow (H_1, H_2)$ says that G can be edge colored with 2 colors such that it contains (as a subgraph) no H_1 in the first and no H_2 in the second color.

Problem 2. Given graphs G, H_1, H_2, determine whether G can be edge colored with 2 colors avoiding monochromatic subgraphs isomorphic to H_1 and H_2 in their respective colors, i.e., determine whether $G \rightarrow (H_1, H_2)$.

Schaefer [114] showed that this problem is co-$\mathcal{NP}^{\mathcal{NP}}$-complete, i.e., complete for the second level of the polynomial hierarchy.

With this insight Haanpää further explains that when restricting G to be a clique, the problem at least remains $\mathcal{NP}$-hard(see a paper of Burr [21] for a proof). When all three input graphs are restricted to be complete graphs, we do not know whether the restricted problem remains $\mathcal{NP}$-hard. We do know that the problem is in co-$\mathcal{NP}^{\mathcal{NP}}$. This is only true if the input graphs are explicitly given as complete graphs, i.e., if the size of the description of a complete graph of size n is polynomial in n. However, when we ask for a Ramsey number $R(k_1, k_2; 2)$, the input is not encoded as a graph (and neither as a unary number), but rather as two numbers k_1 and k_2. Consequently the input format is exponentially smaller than the input format that explicitly provides the graphs. Thus we do not know whether the problem of computing the Ramsey numbers $R(k_1, k_2; 2)$ is in co-$\mathcal{NP}^{\mathcal{NP}}$or where it is situated in the polynomial hierarchy.

Thus, on the one hand, we do not known whether our problem is in co-$\mathcal{NP}^{\mathcal{NP}}$but, on the other hand, for all we know, an explicit formula that computes Ramsey numbers in linear time might exist.

4.6 Previous algorithms

As shown by Figure 4.2 only very few Ramsey numbers are known. Thus the computational methods that compute new Ramsey numbers are also few. Algorithms that compute exact Ramsey numbers are algorithms that show exact upper bounds (rather than lower bounds for which we have short certificates, that are independent of the algorithms they were produced with).

Bounds for Ramsey numbers have been established by reformulating the problem into an integer program. Current solvers for these integer programs have become very fast. However, the computation of the values of $R(3, 8; 2)$ and $R(4, 5; 2)$ did not use these techniques. The value of $R(3, 9; 2)$ was known before the value of $R(3, 8; 2)$ [54]. The computation of $R(3, 9; 2)$ involved some computer support, and combinatorial arguments were used to drastically reduce the search space. The computation of the values of $R(3, 8; 2)$ and $R(4, 5; 2)$ heavily depended on the algorithmic design of the programs used for their computation. We only give a very rough overview of the ingredients of the algorithms used for the computation of both values:

Combinatorial arguments: A combinatorial argument may for example bound the number of edges in an extremal graph or the minimum degree. Arguments of this form played a crucial role in search space reduction, which reduced the running time of the exhaustive enumeration of the Ramsey graphs.

Isomorphism rejection: When dealing with enumeration of graphs of some subclass, isomorphism rejection is crucial. For the computation of $R(3, 8; 2)$ and $R(4, 5; 2)$ Nauty (See Section 2.2) was used, as it is fast and moreover computes canonical labelings. Given the canonical labeling of a graph, isomorphism detection is trivial.

Algorithm engineering: Appropriate data structures (such as bit-vectors) to reduce the required machine instructions for basic operations, as well as design techniques (such as clever enumeration and search space reduction) are required to reduce the global running time.

Gluing: The term gluing describes the method of composing Ramsey graphs from Ramsey graphs of smaller parameters. This is possible, as the proof of Ramsey's theorem shows: Any $(4, 5; 2)$-Ramsey graph decomposes into a vertex v, its neighborhood, which forms a $(3, 5; 2)$-Ramsey graph, and the remaining vertices, which form a $(4, 4; 2)$-Ramsey graph.

All of the ingredients are necessary for the success, i.e., to obtain acceptable running times for the computations of $R(3, 8; 2)$ and $R(4, 5; 2)$. For further details we refer the reader to two papers: McKay and Min [93] describe the methods employed for the computation of $R(3, 8; 2)$, and McKay and Radziszowski [94] describe the methods employed for the computation of $R(4, 5; 2)$.

4.7 The wildcards algorithm for Ramsey numbers

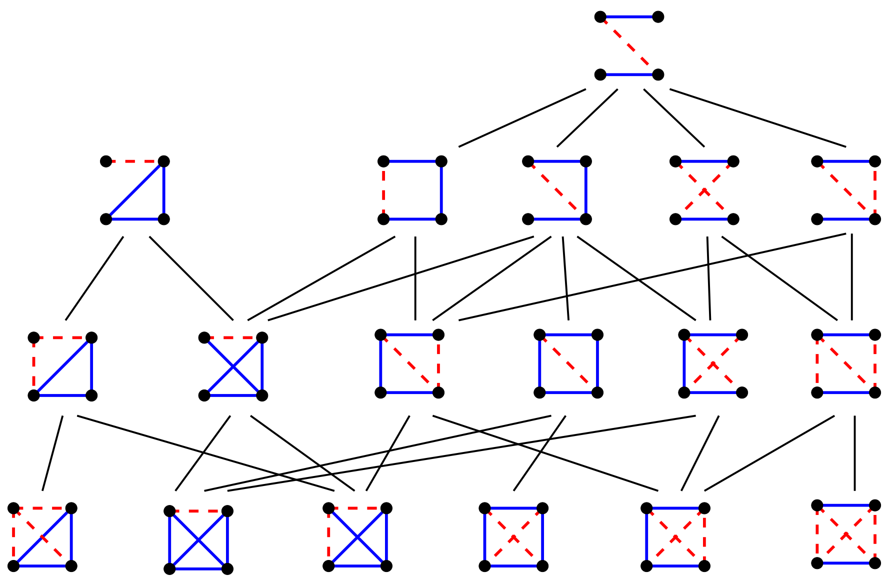

Figure 4.3: The figure depicts the partial order of all varicolored $(3,4;2)$-Ramsey graphs of size 4. The two colors are shown in red (dashed) and blue (solid). Though the graphs are all complete graphs, for improved lucidity the edges colored in the varicolor $\{\text{red}, \text{blue}\}$ are omitted. Following a line downwards corresponds to the specification of an edge from varicolor $\{\text{red}, \text{blue}\}$ to either red or blue.

We now combine the ScrewBox algorithm (see Section 2.6) and the wildcards algorithm (see Section 3.8) designed in the previous chapters to a wildcards algorithm that computes Ramsey numbers.

When dealing with unlabeled graphs, as in the computation of Ramsey numbers, isomorphic copies have to be eliminated in order to avoid an explosion of the search space. In the wildcards algorithm we need to detect isomorphisms of varicolored graphs, and we need the notion of coverings. In analogy to Definition 34, we first define varicolorings of graphs.

Definition 39 (varicoloring of graphs). Given a graph $G = (V, E)$, a map $\lambda\colon E \to \mathcal{P}(\{1, \ldots, c\}) \setminus \{\}$ is said to be a *varicoloring* of the graph G with c colors.

More precisely the map λ is an edge varicoloring, but since we do not use vertex varicolorings, we omit this specification. Since we deal with unlabeled graphs, when defining whether one varicoloring is coarser than another, we allow that an automorphism is applied to the graph prior to a specification:

Definition 40 (coarser, finer, specification). Given a graph $G = (V, E)$, and two varicolorings $\lambda, \lambda'\colon E \to \mathcal{P}(\{1, \ldots, c\}) \setminus \{\}$, we say that λ is *coarser* than λ', (and λ'

is *finer* than λ') if there is an automorphism ϕ of G such that

$$\forall \{v_1, v_2\} \in E\colon\ \lambda'(\{v_1, v_2\}) \subseteq \lambda(\{\phi(v_1), \phi(v_2)\}).$$

In the case where ϕ can be taken as the identity, we also say that λ *specifies* to λ'.

Thus λ is coarser than λ' if λ specifies to a permutation of the λ' colored graph G. We say that λ *covers* λ' if any specification of λ' to an ordinary edge coloring is finer than λ. Extending our previous definition of an ordinarily colored Ramsey graph, we say that a complete varicolored graph is $(k_1, \ldots, k_c; c)$-Ramsey if it does not specify to an ordinary coloring that is not $(k_1, \ldots, k_c; c)$-Ramsey. In this case the varicoloring is *proper*.

The covering relation induces a partial order on the set of varicolorings of a graph G. We are in particular interested in the suborder of Ramsey varicolorings, within the order of varicolorings of a complete graph K_n, for $n \in \mathbb{N}$. For $n = 4$, Figure 4.3 shows this suborder of the $(3, 4; 2)$-Ramsey graphs of size 4.

For the remainder of this section we fix the number of colors $c \in \mathbb{N}$ and the parameters $k_1, \ldots, k_c \in \{1, 2, \ldots\}$.

To compute $R(k_1, \ldots, k_c; c)$ we proceed in a similar fashion as in the previous chapter: For $n \in \{1, \ldots, R(k_1, \ldots, k_c; c)\}$, we iteratively construct a list $\mathcal{L}_n$ of proper varicolorings of K_n. (We fix the vertex set K_n as $V(K_n) = \{1, \ldots, n\}$.) Each $\mathcal{L}_n$ covers all proper ordinary colorings of the complete graph K_n. A list $\mathcal{L}_n$ is empty if and only if $n \geq R(k_1, \ldots, k_c; c)$. We thus construct the lists $\mathcal{L}_1, \mathcal{L}_2, \ldots$, and once we observe that the list $\mathcal{L}_n$ is empty for the first time, we conclude that $n = R(k_1, \ldots, k_c; c)$.

For efficiency it is necessary to keep the maintained lists $\mathcal{L}_1, \mathcal{L}_2, \ldots, \mathcal{L}_{R(k_1,\ldots,k_c;c)}$ small. In particular, we need to avoid treating isomorphic copies of the graphs. This complicates the algorithm in comparison to the wildcards algorithm in the previous chapter. A 2-graph with d unspecified edges does not necessarily cover 2^d ordinarily colored graphs. Note, for example, that though the two maximal graphs in Figure 4.3 (i.e., the graphs at the top) have 2 respectively 3 undetermined edges, the number of proper ordinary colorings, i.e., the number of ordinarily colored $(3, 4; 2)$-Ramsey graphs of size 4 (i.e., the graphs shown at the bottom) covered by each of them is 3 and 5 respectively. Furthermore, since they do not form a downwards antichain, together they do not cover $3 + 5 = 8$ but 6 proper ordinary colorings. In particular we lack an analogue of Lemma 6, concerning downward antichains of mixed van der Waerden numbers, that allows us to find a set $\mathcal{L}_n$ that covers every ordinary coloring exactly once.

We start with a high level description of a backtracking algorithm that performs our desired task of computing Ramsey numbers.

4.7.1 High level description of the wildcards algorithm

Assuming we are given a list $\mathcal{L}_n$ of varicolorings of K_n, we want to construct a list $\mathcal{L}_{n+1}$ of varicolorings of K_{n+1} that covers all ordinary colorings of K_{n+1}. We gradually build the list $\mathcal{L}_{n+1}$. One by one we pick a varicoloring λ from $\mathcal{L}_n$ and extend it by the additional vertex $n+1$, and color all new edges with $C := \{1,\dots,c\}$, i.e., we form the varicoloring $\lambda_{(n+1)\to C}$ given by

$$\lambda_{(n+1)\to C}(e) := \begin{cases} \lambda(e) & \text{if } (n+1) \notin e, \\ C & \text{otherwise.} \end{cases}$$

This varicoloring $\lambda_{(n+1)\to C}$ is in general not proper, thus, in a backtracking fashion, we specify edges until the varicoloring is proper. We then test whether the coloring is already covered by a graph in $\mathcal{L}_{n+1}$. If it is, we discard it and backtrack (as explained below, i.e., we continue with a different varicoloring from the list $\mathcal{L}_n$). If it is not, we have to include the current varicoloring (or a varicoloring coarser than the current varicoloring) into the list $\mathcal{L}_{n+1}$. Any varicoloring coarser than the current varicoloring can be used to guarantee that the current varicoloring is covered. Hence, when choosing the varicoloring that is inserted into the list $\mathcal{L}_{n+1}$, various options exist. Our option of choice is to insert some maximal proper varicoloring coarser than the current varicoloring into the list $\mathcal{L}_{n+1}$.

We now explain how to perform the actual backtracking. It proceeds the same way as the backtracking from Section 3.8 proceeds. Figure 4.4 describes this backtracking procedure for two colors. We adopt the terminology for colors to be able to describe the backtracking:

Given a graph G and a varicoloring $\lambda\colon E(G) \to \mathcal{P}(\{1,\dots,c\}) \setminus \{\}$, we define for edge $e \in E(G)$ and varicolor $T \subseteq \{1,\dots,c\}$ the *recoloring of λ of edge e with color T* as the varicoloring $\lambda_{e\to T}\colon E(G) \to \mathcal{P}(\{1,\dots,c\}) \setminus \{\}$ given by

$$\lambda_{e\to T}(e') := \begin{cases} \lambda(e') & \text{if } e' \neq e, \\ T & \text{if } e' = e. \end{cases}$$

Definition 41 (prohibited, innocuous). Let $\lambda\colon E(G) \to \mathcal{P}(\{1,\dots,c\}) \setminus \{\}$ be a varicoloring of a graph G. Let $t \in \{1,\dots,c\}$ be an ordinary color, $e \in E(G)$ an edge, and $\lambda_{e\to\{t\}}$ the recoloring of edge e with color $\{t\}$.

- We say t is *prohibited* for edge e if $\lambda_{e\to\{t\}}$ contains a monochromatic clique of color $\{t\}$ that contains the edge e.
- We say t is *innocuous* for edge e if $\lambda_{e\to\{t\}}$ does not specify to any coloring which contains a monochromatic clique of color $\{t\}$ that contains the edge e.

For a varicoloring λ from $\mathcal{L}_n$, the backtracking starts with $\lambda' = \lambda_{(n+1)\to C}$, the extension of λ by an additional vertex. We backtrack the following way (compare with Section 3.8):

First, while there is an $e \in E(G)$ whose varicolor $T = \lambda'(e)$ contains a prohibited color $t \in T$, we remove that color t from the varicolor of e, i.e., we form the varicoloring $\lambda'_{e\to T\setminus\{t\}}$. This step eliminates unnecessary braching. We now suppose all edges in λ' are colored with non-prohibited colors. If λ' is not proper, then there exists an

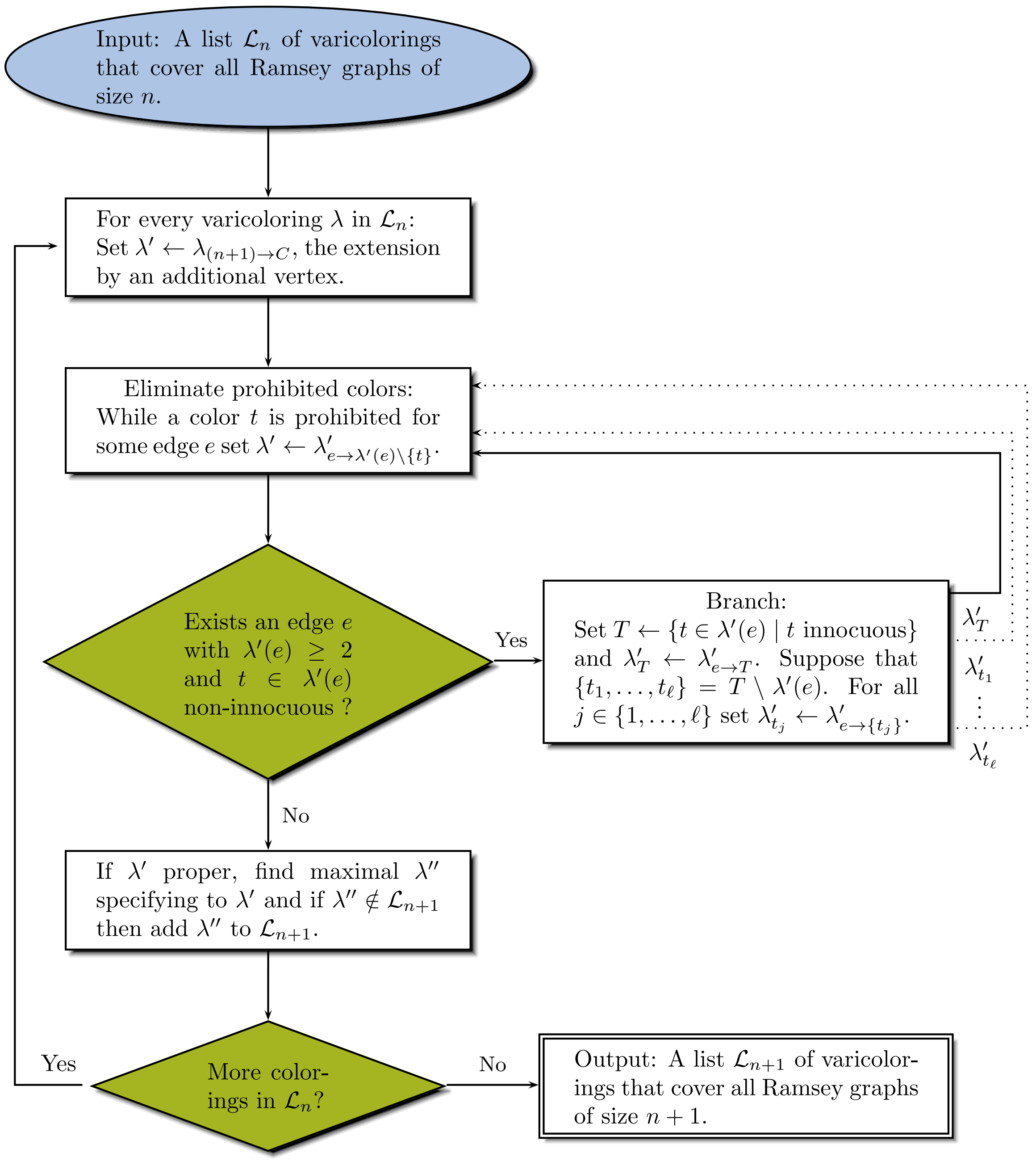
Input: A list $\mathcal{L}_n$ of varicolorings that cover all Ramsey graphs of size n.
For every varicoloring λ in $\mathcal{L}_n$: Set $\lambda' \leftarrow \lambda_{(n+1)\to C}$, the extension by an additional vertex.
Eliminate prohibited colors: While a color t is prohibited for some edge e set $\lambda' \leftarrow \lambda'_{e\to\lambda'(e)\setminus\{t\}}$.
Exists an edge e with $\lambda'(e) \geq 2$ and $t \in \lambda'(e)$ non-innocuous ?
Yes
Branch: Set $T \leftarrow \{t \in \lambda'(e) \mid t \text{ innocuous}\}$ and $\lambda'_T \leftarrow \lambda'_{e\to T}$. Suppose that $\{t_1,\ldots,t_\ell\} = T \setminus \lambda'(e)$. For all $j \in \{1,\ldots,\ell\}$ set $\lambda'_{t_j} \leftarrow \lambda'_{e\to\{t_j\}}$.
λ'_T
λ'_{t_1}
$\vdots$
λ'_{t_ℓ}
No
If λ' proper, find maximal λ'' specifying to λ' and if $\lambda'' \notin \mathcal{L}_{n+1}$ then add λ'' to $\mathcal{L}_{n+1}$.
Yes
More colorings in $\mathcal{L}_n$?
No
Output: A list $\mathcal{L}_{n+1}$ of varicolorings that cover all Ramsey graphs of size $n+1$.

Figure 4.4: The figure shows a high level view of the wildcards algorithms for Ramsey numbers. Given a list $\mathcal{L}_n$ of varicolorings of size n, it produces a list $\mathcal{L}_{n+1}$ of varicolorings of size $n+1$.

edge e, whose varicolor $T = \lambda'(e)$ contains at least two ordinary colors and which also contains a color that is not innocuous. We construct the set $\Lambda :=$

$$\{\lambda'_{e \to T'}\} \cup \{\lambda'_{e \to \{t\}} \mid t \in T \setminus T'\},$$

on which we recurse (i.e., we perform the backtracking with all colorings in Λ) until they are proper.

Remaining to be explain is what we do once we obtain a proper coloring λ' say. When the current varicoloring λ' is proper, we heuristically check whether the varicoloring is already covered by a graph in $\mathcal{L}_{n+1}$: We find a maximal proper varicoloring λ'' that specifies to λ'. We then check whether there is an isomorphic varicoloring already in the list $\mathcal{L}_{n+1}$, i.e., we check whether the complete graph K_{n+1} colored with λ'' is isomorphic to the graph K_{n+1} colored with some coloring from $\mathcal{L}_{n+1}$. If this is the case, we discard λ', otherwise we insert λ'' into the list, which also covers λ'.

As we said, this check is only heuristic. For the problem of detecting whether a graph in $\mathcal{L}_{n+1}$ already covers a varicoloring λ', we do not expect to find an efficient solution. The reason is that the $\mathcal{NP}$-hard problem MAX-CLIQUE reduces to this problem. We use the specified heuristic since (supposedly) the set of maximal proper varicolorings is small.

For this heuristic check, we require an isomorphism test. For our purposes, it is not essential that an isomorphic copy always be found, if one exists. On the one hand, to keep the list $\mathcal{L}_{n+1}$ small, we only need that for most graphs an isomorphic copy is found. On the other hand, we must ensure that we do not err when choosing not to include a graph, i.e., we may not discard a graph that is not covered by another graph in the list. This functionality is especially offered by the ScrewBox algorithm (see Section 2.6).

To be able to screen a single graph against a large library (in our case the list $\mathcal{L}_{n+1}$), we have to modify the ScrewBox algorithm. We use easy invariants that differentiate most graphs for a preselection of the graphs. Only on the remaining few graphs, for which these invariants do not suffice, we resort to the ScrewBox to test for isomorphism. Note that if, for a specific class of isomorphic graphs, we do not perform isomorphism testing at all, or we never find the graphs to be isomorphic, the list may contain an exponential number of isomorphic copies of the graph, thus, for any specific graph, we need to detect isomorphism most of the time.

During the algorithm, by assumption, deletion of the last vertex $n + 1$ gives us a proper varicoloring of the graph. Therefore, in order to check whether a color is prohibited or innocuous, we only need to consider cliques that contain the last vertex. We thereby save computation time.

This concludes our description of the backtracking procedure. We now explain how gluing may be performed with varicolorings and how it helps to increase the efficiency of the algorithm.

4.7.2 The Gluing technique for the wildcards algorithm

Most techniques that use combinatorial arguments (see Section 4.6) can not straightforwardly be used on varicolored graphs. An example of this is a restriction on the maximum degree D_t in a specific color t, i.e., we require that any vertex is incident to at most D_t edges of color t. It is not clear how to impose this restriction onto varicolorings. If λ is a varicoloring, then the number of edges varicolored in $\{t\}$ which are incident to a fixed vertex varies among specifications of λ to ordinary colorings. In this case the coloring may cover ordinary colorings that fulfill the degree constraint and at the same time cover ordinary colorings that do not fulfill the constraint.

In contrast to the technique that restricts degrees, the technique of gluing may be directly applied to varicolored objects. The graph obtained by gluing two disjoint varicolored graphs $G = (V, E)$ and $H = (V', E')$, varicolored with λ_G and λ_H with colors from the set C, is the join $G * H = (V \cup V', E \cup E' \cup \{\{v_g, v_h\} \mid v_g \in G, v_h \in H\})$, with the varicoloring λ_{G*H}, given by

$$\lambda_{G*H}(e) := \begin{cases} \lambda_G(e) & \text{if } e \in E(G), \\ \lambda_H(e) & \text{if } e \in E(H), \\ C & \text{otherwise.} \end{cases}$$

I.e., in the join every edge that is either completely contained in G or completely contained in H retains its color, and all other edges are colored with the whole set of colors C.

The coloring of a graph we obtain by this gluing procedure is not necessarily proper. We thus have to specify the coloring further. For this we use the backtracking method described in the previous subsection.

The advantage of the gluing technique is the applicability of the decomposition in Ramsey's theorem. For example, a $(k_1, k_2; 2)$-Ramsey graph decomposes into a vertex, a $(k_1-1, k_2; 2)$-Ramsey graph and a $(k_1, k_2-1; 2)$-Ramsey graph (see Figure 4.1). Since the smaller parameters are more restrictive, there are less graphs to these parameters, so the gluing allows us to reduce the search space.

When the gluing technique and the subsequent backtracking are performed with two ordinarily colored graphs, this approach is essentially the interval technique used in [94] for the computation of $R(4, 5; 2)$.

4.8 Certification

Mostly everything we said about certifying mixed van der Waerden numbers in Section 3.11 caries over to the certification of Ramsey number computation. Lower bounds can easily be certified by providing Ramsey graphs. McKay has gathered an extensive pool of Ramsey graphs, including extremal graphs for all 2-color Ramsey numbers whose determination required computational power [90]. The validity of these graphs can be checked with any MAX-CLIQUE algorithm. As the graphs are relatively small with only small cliques and small independent sets, this computation is feasible. However, again, there seems to be no satisfying way to provide certified upper bounds, unless they are of combinatorial nature obtained without computation. Further, it is unclear how or if we can exploit randomization to certify upper bounds. (Compare with Subsection 2.11.1.)

4.9 Evaluation and conclusion

To evaluate the approach taken by the wildcards algorithm for Ramsey numbers, we first, as for the case of mixed van der Waerden numbers, compare the number of Ramsey graphs with the size of the lists of varicolored Ramsey graphs that are produced by the wildcards algorithm. Analyzing these numbers, we observe a similar behavior as in the previous chapter. For various parameters, Figures 4.5–4.8 show the number of Ramsey graphs and the sizes the lists $\mathcal{L}_n$ of varicolored Ramsey graphs produced with the wildcards algorithm (for the same parameters). We observe that there is a peak in the function that describes the numbers of non-isomorphic Ramsey graphs to a given size, and the function drastically decreases again after the peak. Note that all sequences of numbers are unimodal (i.e, they increase up to some point and then decrease again). Intuitively this is what we expect: For small sizes the number of Ramsey graphs increases. This is due to the fact that the number of graphs increases and most of the graphs are Ramsey graphs. At some point the Ramsey property of the graphs, i.e., the fact that monochromatic cliques must be avoided, starts to force more and more structure onto the graphs, until the Ramsey property cannot be fulfilled anymore. Maybe for all parameters, the number of non-isomorphic Ramsey graphs shows this behavior:

Open Question 2. Is the number of non-isomorphic $(k_1, \ldots, k_c; c)$-Ramsey graphs of size n, for fixed parameters $c \in \mathbb{N}$ and $k_1, \ldots, k_c \in \{1, 2, \ldots\}$, a unimodal function in n?

We also observe that the number of the lists $\mathcal{L}_n$ in the range of the peak is far less than the number of ordinarily colored graphs. There are, for example, 275086 non-isomorphic $(3, 6; 2)$-Ramsey graphs of size 14, whereas only 1479 varicolored graphs are contained in the list $\mathcal{L}_{14}$ produced by the wildcards algorithm. By simultaneously considering proper ordinary colorings of Ramsey graphs, the varicoloring approach thus yields a search space contraction by providing a small list of varicolorings that contains the essential information on all proper ordinary colorings. The gluing operation uses two of these lists of varicolorings to compose a new list of larger varicolorings that again contains all essential information.

With the wildcards algorithm, the list $\mathcal{L}_{22}$ of extremal $(3, 7; 2)$-Ramsey graphs of size 22 has been computed. Recall that $R(3, 7; 2) = 23$. (Also the list $\mathcal{L}_{21}$ has been computed. The computation of the list $\mathcal{L}_n$ for $n < 21$ was avoided with the help of the gluing operation). Using the list $\mathcal{L}_{22}$, all $(3, 7; 2)$-Ramsey graphs of size 22 have been computed. There are 191 such graphs (which has previously been known [90]). For the

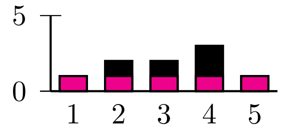

Figure 4.5: Number of $(3, 3; 2)$-Ramsey graphs

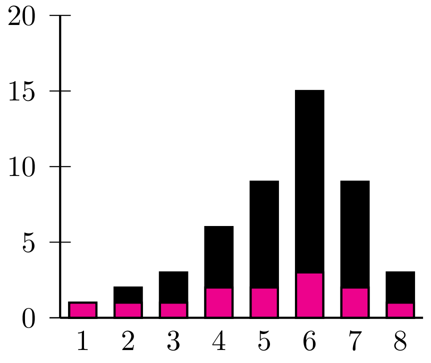

Figure 4.6: Number of $(3, 4; 2)$-Ramsey graphs

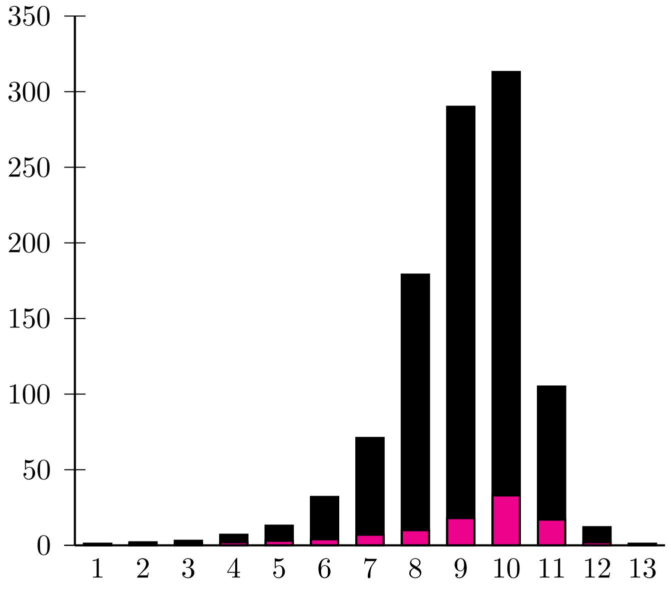

Figure 4.7: Number of $(3, 5; 2)$-Ramsey graphs

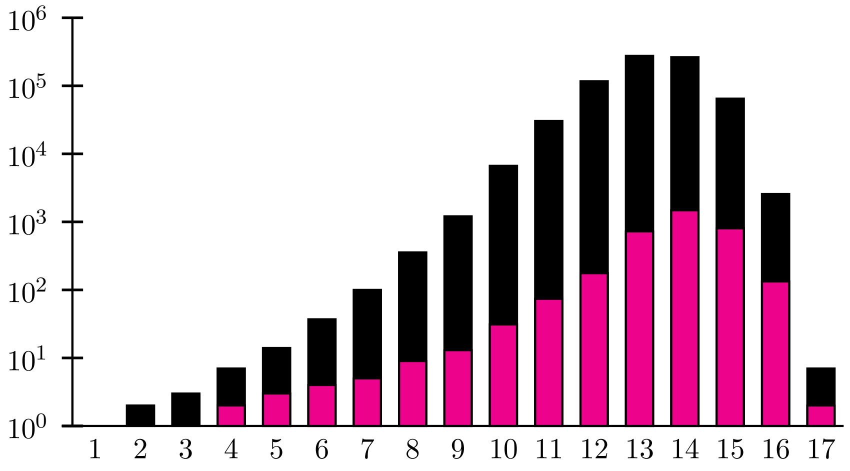

Figure 4.8: Number of $(3, 6; 2)$-Ramsey graphs

The figures show the number of $(3, k_2; 2)$-Ramsey graphs for $k_2 \in \{3, 4, 5, 6\}$, and the sizes of the lists $\mathcal{L}_n$ of maximal proper varicolorings that are produced by the wildcards algorithm. The elements of the lists cover all Ramsey graphs of the respective size and parameters. Note that Figure 4.8 is shown in logarithmic scale.

computation of all $(3, 8; 2)$-Ramsey graphs of extremal size 27 (which has previously not been done [90]), the efficiency of the algorithm is not sufficient.

Three major challenges remain to improve the efficiency of the implementation of the wildcards algorithm: First, the enumeration of the varicolorings produces an overhead to cope with the ambiguity intrinsic in the approach. This overhead computes monochromatic cliques, for which an efficient algorithm is required. Second, though the isomorphism question is equally complicated for ordinary colors as for varicolors, the relation of two graphs in the partial order arising from varicolorings is difficult to determine. And third, the space consumption of the algorithm is of concern. When we compute Ramsey numbers for larger parameters, the space required to maintain the lists $\mathcal{L}_n$ exceeds the space available in the main memory of current machines. For the mixed van der Waerden numbers, there is no need to generate the lists explicitly since no duplicates appear. In other words, while a depth-first strategy with linear space requirement is performed for the computation of the van der Waerden numbers, for the computation of the Ramsey numbers we must employ a breadth-first strategy, which results in the large space requirement. Along with the search space contraction that is achieved by the wildcards algorithm, we thus further require efficient implementations and clever enumeration techniques to compute the next Ramsey number.

Overall, we conclude that computing Ramsey numbers is still a difficult challenge, and remains science fiction for the time being, just as in the famous quote by Erdős himself [52]:

"Aliens invade the earth and threaten to obliterate it in a year's time unless human beings can find the Ramsey number for red five and blue five. We could marshal the world's best minds and fastest computers, and within a year we could probably calculate the value. If the aliens demanded the Ramsey number for red six and blue six, however, we would have no choice but to launch a preemptive attack."

List of Algorithms

List of Figures

Bibliography

[1] Tanbir Ahmed. van der Waerden numbers. http://users.encs.concordia.ca/∼ta_ahmed/vdw.html. 98, 113

[2] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. Wiley-Interscience, August 2000. 94

[3] László Babai. Moderately exponential bound for graph isomorphism. In *FCT '81: Proceedings of the 1981 International FCT-Conference on Fundamentals of Computation Theory*, pages 34–50, London, UK, 1981. Springer-Verlag. 21, 30

[4] László Babai. *Handbook of Combinatorics (vol. 2)*, chapter Automorphism groups, isomorphism, reconstruction, pages 1447–1540. MIT Press, Cambridge, MA, USA, 1995. 16

[5] László Babai, Paul Erdős, and Stanley M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9(3):628–635, 1980. 20

[6] László Babai, D. Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 310–324, New York, NY, USA, 1982. ACM. 20

[7] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *STOC '83: Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183, New York, NY, USA, 1983. ACM. 31

[8] David A. Basin. A term equality problem equivalent to graph isomorphism. *Information Processing Letters*, 51(2):61–66, 1994. 19

[9] Burak Bayoglu and Ibrahim Sogukpinar. Polymorphic worm detection using token-pair signatures. In *SecPerU '08: Proceedings of the fourth international workshop on Security, privacy and trust in pervasive and ubiquitous computing*, pages 7–12, New York, NY, USA, 2008. ACM. 11

[10] Michael D. Beeler. A new van der Waerden number. *Discrete Applied Mathematics*, 6(2):207–207, 1983. 98

[11] Michael D. Beeler and Patrick E. O'Neil. Some new van der Waerden numbers. *Discrete Mathematics*, 28(2):135–146, 1979. 98, 99

[12] Elwyn Ralph Berlekamp. A construction for partitions which avoid long arithmetic progressions. *Canadian Mathematical Bulletin*, 11(3):409–414, 1968. 96

[13] Hans L. Bodlaender. Polynomial algorithms for graph isomorphism and chromatic index on partial k-trees. *Journal of Algorithms*, 11(4):631–643, 1990. 20

[14] Tom Bohman. The triangle-free process. `arXiv:0806.4375v1 [math.CO]`, 2008. 118

[15] Kellogg S. Booth. Isomorphism testing for graphs, semigroups, and finite automata are polynomially equivalent problems. *SIAM Journal on Computing*, 7(3):273–279, 1978. 19

[16] Ravi B. Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. In *SWAT'90: Proceedings of the second Scandinavian Workshop on Algorithm Theory*, pages 13–25, Berlin, Germany, 1990. Springer-Verlag. 11

[17] Ravi B. Boppana and Magnús M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. *BIT Numerical Mathematics*, 32(2):180–196, 1992. 11

[18] Andries E. Brouwer. *Handbook of Combinatorics (vol. 1)*, chapter Block designs, pages 693–745. MIT Press, Cambridge, MA, USA, 1995. 63

[19] Tom C. Brown. Some new van der Waerden numbers (preliminary report). *Notices of the American Mathematical Society*, 21:A–432, 1974. 98

[20] Tom C. Brown, Bruce M. Landman, and Aaron Robertson. Bounds on some van der Waerden numbers. *Journal of Combinatorial Theory, Series A*, 115(7):1304–1309, 2008. 96

[21] Stefan A. Burr. *Mathematics of Ramsey Theory*, chapter On the computational complexity of ramsey-type problems. Springer-Verlag, New York, NY, USA, 1990. 119

[22] Jin-Yi Cai. From the world of P and NP: Problems in complexity theory. `http://www.cs.wisc.edu/~jyc/MadMath.pdf`. 87

[23] Jin-Yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, 12(4):389–410, 1992. 26, 29, 39

[24] Peter J. Cameron. 6-transitive graphs. *Journal of Combinatorial Theory, Series B*, 28(2):168–179, 1980. 61

[25] Peter J. Cameron. *Topics in Algebraic Graph Theory*, chapter Strongly regular graphs, pages 203–221. Cambridge University Press, New York, NY, USA, 2004. 61

[26] Vašek Chvátal. Some unknown van der Waerden numbers. In *Proceedings of the Calgary International Conference on Combinatorial Structures and Their Applications*, pages 31–33, New York - London - Paris, 1970. Gordon and Breach. 98

[27] Marlene J. Colbourn and Charles J. Colbourn. Concerning the complexity of deciding isomorphism of block designs. *Discrete Applied Mathematics*, 3(3):155–162, 1981. 19, 20

[28] David Conlon. A new upper bound for diagonal Ramsey numbers. *Annals of Mathematics*, to appear, 2009. 117

[29] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990. 83

[30] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. Wiley and Sons, New York, August 1991. 53

[31] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for cnf. In *DAC'04: Proceedings of the 41th Design Automation Conference*, pages 530–534, New York, NY, USA, June 2004. ACM. 24

[32] Matt DeVos and Robert Šámal. The Open Problem Garden. `http://garden.irmacs.sfu.ca/`. 118

[33] Michael R. Dransfield, Lengning Liu, Victor W. Marek, and Miroslaw Truszczynski. Satisfiability and computing van der Waerden numbers. *Electronic Journal of Combinatorics*, 11(1):R41, 2004. 11, 100

[34] Carl Droms. Isomorphisms of graph groups. *Proceedings of the American Mathematical Society*, 100(3):407–408, 1987. 19

[35] Carl Ebeling. GeminiII: a second generation layout validation program. In *ICCAD'88: IEEE International Conference on Computer-Aided Design*, pages 322–325, Washington, DC, USA, Nov 1988. IEEE Computer Society. 10

[36] Paul Erdős and Lászlo Lovász. Problems and results on 3-chromatic hypergraphs and some related questions. In *Infinite and Finite Sets (Colloquia Mathematica Societatis Jbanos Bolyai 11)*, pages 609–627, 1975. 95

[37] Paul Erdős and Richard Rado. Combinatorial theorems on classifications of subsets of a given set. *Proceedings of the London Mathematical Society*, s3-2(1):417–439, 1952. 94

[38] Jeff Erickson. Finding longest arithmetic progressions. `http://www.cs.uiuc.edu/~jeffe/pubs/arith.html`. 98

[39] Leonhard Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii Academiae Scientarum Imperialis Petropolitanae*, 8:128–140, 1736. 9

[40] Joan Feigenbaum and Alejandro A. Schäffer. Recognizing composite graphs is equivalent to testing graph isomorphism. *SIAM Journal on Computing*, 15(2):619–627, 1986. 20

[41] Ion S. Filotti and Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 236–243, New York, NY, USA, 1980. ACM. 20

[42] Martin Fürer. A counterexample in graph isomorphism testing. Technical Report CS-87-36, Pennsylvania State University, Department of Computer Science, University Park, PA, USA, 1987. 26

[43] Martin Fürer. Graph isomorphism testing without numerics for graphs of bounded eigenvalue multiplicity. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on discrete algorithms*, pages 624–631, San Francisco, CA, USA, 1995. SIAM. 25

[44] Merrick Furst, John Hopcroft, and Eugene Luks. Polynomial-time algorithms for permutation groups. In *FOCS '80: Proceedings of the twenty-first Annual Symposium on Foundations of Computer Science*, pages 36–41, Washington, DC, USA, 1980. IEEE Computer Society. 20

[45] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979. 15

[46] Max Garzon and Yechezkel Zalcstein. The complexity of isomorphism testing. In *FOCS '86: Proceedings of the twenty-seventh Annual Symposium on Foundations of Computer Science*, pages 313–321, Washington, DC, USA, 1986. IEEE Computer Society. 19

[47] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38(3):690–728, 1991. 18

[48] William Timothy Gowers. The two cultures of mathematics. `http://www.dpmms.cam.ac.uk/~wtg10/2cultures.ps`. 11

[49] William Timothy Gowers. A new proof of Szemerédi's theorem. *Geometric And Functional Analysis*, 11(3):465–588, 2001. 94

[50] Ronald L. Graham and Bruce L. Rothschild. A short proof of van der Waerden's theorem on arithmetic progressions. *Proceedings of the American Mathematical Society*, 42(2):385–386, 1974. 91

[51] Ronald L. Graham, Bruce L. Rothschild, and Joel H. Spencer. *Ramsey Theory*. John Wiley & Sons, Inc., New York, NY, USA, 1990. 90, 91, 94, 96, 117, 118

[52] Ronald L. Graham and Joel H. Spencer. Ramsey theory. *Scientific American*, 262(7):112–117, 1990. 129

[53] Ben Green and Terence Tao. The primes contain arbitrarily long arithmetic progressions. *Annals of Mathematics*, 167(2):481–547, 2008. 89

[54] Charles M Grinstead and Sam M Roberts. On the Ramsey numbers $R(3,8)$ and $R(3,9)$. *Journal of Combinatorial Theory, Series B*, 33(1):27–51, 1982. 120

[55] Martin Grohe. Isomorphism testing for embeddable graphs through definability. In *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 63–72, New York, NY, USA, 2000. ACM. 25

[56] Martin Grohe and Julian Mariño. Definability and descriptive complexity on databases of bounded tree-width. In *ICDT '99: Proceedings of the seventh International Conference on Database Theory*, pages 70–82, London, UK, 1999. Springer-Verlag. 25

[57] Harri Haanpää. *Computational Methods for Ramsey Numbers*. Licentiate thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, June 2000. 119

[58] Harri Haanpää. Computational methods for Ramsey numbers. Reseach Report A65, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, November 2000. 119

[59] Torben Hagerup and Christine Rüb. A guided tour of chernoff bounds. *Information Processing Letters*, 33(6):305–308, 1990. 80

[60] Edith Hemaspaandra, Lane A. Hemaspaandra, Stanislaw P. Radziszowski, and Rahul Tripathi. Complexity results in graph reconstruction. *Discrete Applied Mathematics*, 155(2):103–118, 2007. 29th Symposium on Mathematical Foundations of Computer Science MFCS 2004. 20

[61] Paul R. Herwig, Marijn J. H. Heule, P. Martijn van Lambalgen, and Hans van Maaren. A new method to construct lower bounds for van der Waerden numbers. *Electronic Journal of Combinatorics*, 14(1):R6, 2007. 111

[62] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 310–324, New York, NY, USA, 1974. ACM. 20

[63] Neil Immerman and Eric Lander. *Complexity Theory Retrospective*, chapter Describing Graphs: A First-Order Approach to Graph Canonization. Springer-Verlag, Berlin, Germany, 1990. 25

[64] T. S. Jayram, Ravi Kumar, and D. Sivakumar. Two applications of information complexity. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 673–682, New York, NY, USA, 2003. ACM. 85

[65] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *ALENEX'07: Proceedings of the ninth Workshop on Algorithm Engineering and Experiments*, pages 135–149, New Orleans, USA, 2007. SIAM. 24, 71, 76, 77, 78

[66] Volker Kaibel and Alexander Schwartz. On the complexity of polytope isomorphism problems. *Graphs and Combinatorics*, 19(2):215–230, 2003. 19

[67] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000. 84

[68] David R. Karger and Debmalya Panigrahi. A near-linear time algorithm for constructing a cactus representation of minimum cuts. In *SODA '09: Proceedings of the nineteenth annual ACM-SIAM symposium on discrete algorithms*, pages 246–255, Philadelphia, PA, USA, 2009. SIAM. 84

[69] Richard M. Karp and Robert Kleinberg. Noisy binary search and its applications. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms*, pages 881–890, Philadelphia, PA, USA, 2007. SIAM. 51, 52

[70] Jeong Han Kim. The ramsey number $R(3,t)$ has order of magnitude $t^2/\log t$. *Random Structures and Algorithms*, 7(3):173–207, 1995. 118

[71] Tracy Kimbrel and Rakesh Kumar Sinha. A probabilistic algorithm for verifying matrix products using $o(n^2)$ time and $\log_2 n + o(1)$ random bits. *Information Processing Letters*, 45(2):107–110, 1993. 83

[72] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity.* Birkhäuser Verlag, Basel, Switzerland, Switzerland, 1993. 10, 15, 18, 84

[73] Michal Kouril. *A backtracking framework for beowulf clusters with an extension to multi-cluster computation and sat benchmark problem implementation.* PhD thesis, University of Cincinnati, Cincinnati, OH, USA, 2006. 98, 100, 114

[74] Michal Kouril and Jerome L. Paul. The van der Waerden number $W(2,6)$ is 1132. *Experimental Mathematics*, 17(1):53–61, 2008. 98, 100

[75] Dexter Campbell Kozen. *Complexity of finitely presented algebras.* PhD thesis, Cornell University, Ithaca, NY, USA, 1977. 19

[76] Dexter Campbell Kozen. A clique problem equivalent to graph isomorphism. *SIGACT News*, 10(2):50–52, 1978. 19

[77] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:49–86, 1951. 53

[78] Martin Kutz. *The Angel Problem, Positional Games, and Digraph Roots*. PhD thesis, Freie Universität Berlin, Berlin, Germany, 2004. 19

[79] Martin Kutz and Pascal Schweitzer. ScrewBox: a randomized certifying graph non-isomorphism algorithm. In *ALENEX'07: Proceedings of the ninth Workshop on Algorithm Engineering and Experiments*, pages 150–157, New Orleans, USA, 2007. Society for Industrial and Applied Mathematics, SIAM. 31, 39, 72

[80] Bruce Landman, Aaron Robertson, and Clay Culver. Some new exact van der Waerden numbers. *Integers*, 5(2):A10, 2005. 96, 98

[81] Bruce M. Landman and Aaron Robertson. *Ramsey Theory on the Integers*. American Mathematical Society, February 2004. 90, 99, 100

[82] Jeffrey S. Leon. An algorithm for computing the automorphism group of a Hadamard matrix. *Journal of Combinatorial Theory, Series A*, 27(3):289–306, 1979. 20

[83] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications (2nd ed.)*. Springer-Verlag, Berlin, Germany, 1997. 95

[84] David Lichtenstein. Isomorphism for graphs embeddable on the projective plane. In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 218–224, New York, NY, USA, 1980. ACM. 20

[85] Anna Lubiw. Some NP-complete problems similar to graph isomorphism. *SIAM Journal on Computing*, 10(1):11–21, 1981. 19

[86] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982. 20, 30

[87] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979. 19

[88] Brendan D. McKay. The nauty page.
`http://cs.anu.edu.au/~bdm/nauty`. 11, 15, 18

[89] Brendan D. McKay. Nauty user guide.
`http://cs.anu.edu.au/~bdm/nauty/nug-2.4b7.pdf`. 22, 68

[90] Brendan D. McKay. Ramsey graphs.
`http://cs.anu.edu.au/~bdm/data/ramsey.html`. 126, 127, 129

[91] Brendan D. McKay. Hadamard equivalence via graph isomorphism. *Discrete Mathematics*, 27(2):213–214, 1979. 62

[92] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981. 11, 15, 22, 68

[93] Brendan D. McKay and Zhang Ke Min. The value of the Ramsey number $R(3,8)$. *Journal of Graph Theory*, 16(1):99–105, 1992. 10, 120

[94] Brendan D. McKay and Stanisław P. Radziszowski. $R(4,5) = 25$. *Journal of Graph Theory*, 19(3):309–322, 1995. 10, 120, 126

[95] Kurt Mehlhorn and Stefan Näher. From algorithms to working programs: On the use of program checking in LEDA. In *MFCS '98: Proceedings of the twenty-third International Symposium on Mathematical Foundations of Computer Science*, pages 84–93, London, UK, 1998. Springer-Verlag. 78

[96] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999. 79

[97] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer-Verlag, Berlin, Germany, June 2009. 79

[98] Gary L. Miller. Graph isomorphism, general remarks. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 143–150, New York, NY, USA, 1977. ACM. 19, 20, 21

[99] Gary L. Miller. On the $n \log n$ isomorphism technique (a preliminary report). In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 51–58, New York, NY, USA, 1978. ACM. 20

[100] Gary L. Miller. Isomorphism testing for graphs of bounded genus. In *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 225–235, New York, NY, USA, 1980. ACM. 20

[101] Gary L. Miller. Isomorphism testing and canonical forms for k-contractable graphs (a generalization of bounded valence and bounded genus). In *FCT '83: Proceedings of the 1983 International FCT-Conference on Fundamentals of Computation Theory*, pages 310–327, London, UK, 1983. Springer-Verlag. 31

[102] Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and computation, II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 239–256. American Mathematical Society, 1995. 26, 29, 30

[103] Cristopher Moore, Alexander Russell, and Piotr Sniady. On the impossibility of a quantum sieve algorithm for graph isomorphism. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 536–545, New York, NY, USA, 2007. ACM. 10

[104] Eric Moorhouse. Projective planes of order 27.
`http://www.uwyo.edu/moorhouse/pub/planes27/`. 63, 73

[105] Robin A. Moser. A constructive proof of the Lovasz local lemma. In *STOC '09: Proceedings of the fourty-first annual ACM symposium on Theory of computing*, page to appear, New York, NY, USA, 2009. ACM. 96

[106] Adolfo Piperno. Search space contraction in canonical labeling of graphs (preliminary version). `arXiv:0804.4881v1 [cs.DS]`, 2008. 24

[107] I. N. Ponomarenko. The isomorphism problem for classes of graphs closed under contraction. *Journal of Mathematical Sciences*, 55(2):1621–1643, 1991. 20

[108] William Press, Saul Teukolsky, William Vetterling, and Brian Flannery. *Numerical Recipes in C*. Cambridge University Press, Cambridge, UK, 2nd edition, 1992. 57

[109] Stanisław P. Radziszowski. Small Ramsey numbers. *Electronic Journal of Combinatorics*, page DS1, 2006. 118, 119

[110] Frank P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2-30(1):264–286, 1930. 11, 115, 116

[111] 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. 10

[112] Vera Rosta. Ramsey theory applications. *Electronic Journal of Combinatorics*, page DS13, 2004. 10

[113] Gordon Royle. Projective planes of order 16. `http://www.csse.uwa.edu.au/~gordon/remote/planes16/`. 63, 73

[114] Marcus Schaefer. Graph Ramsey theory and the polynomial hierarchy. In *STOC '99: Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 592–601, New York, NY, USA, 1999. ACM. 119

[115] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312–323, 1988. 18

[116] Pascal Schweitzer. The implementaion of the algorithms developed in this thesis. `http://www.mpi-inf.mpg.de/~pascal/software/`. 13, 41, 111, 112

[117] Pascal Schweitzer. Using the incompressibility method to obtain local lemma results for Ramsey-type problems. *Information Processing Letters*, 109(4):229–232, 2009. 96

[118] Saharon Shelah. Primitive recursive bounds for van der Waerden numbers. *Journal of the American Mathematical Society*, 1(3):683–697, 1988. 94

[119] Joel H. Spencer. Ramsey's theorem - a new lower bound. *Journal of Combinatorial Theory, Series A*, 18(1):108–115, 1975. 117

[120] Joel H. Spencer. *The Strange Logic of Random Graphs*. Springer-Verlag, Berlin, Germany, 2001. 39

[121] R. S. Stevens and R. Shantaram. Computer-generated van der Waerden partitions. *Mathematics of Computation*, 32(142):635–636, 1978. 98

[122] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997. 84

[123] Zoltán Szabó. An application of Lovász's local lemma - a new lower bound for the van der Waerden number. *Random Structures and Algorithms*, 1(3):343–360, 1990. 96

[124] Endre Szemerédi. On sets of integers containing no k elements in arithmetic progression. *Acta Arithmetica*, 27:199–245, 1975. 89

[125] Robert Endre Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1(1):32–34, 1971. 20

[126] Bartel L. van der Waerden. Beweis einer Baudetschen Vermutung. *Nieuw Archief voor Wiskunde*, 15:212–216, 1927. 10, 89, 90

[127] Abraham Wald. Sequential tests of statistical hypotheses. *Annals of Mathematical Statistics*, 16(2):117–186, 1945. 50

[128] Abraham Wald and Jacob Wolfowitz. Optimum character of the sequential probability ratio test. *Annals of Mathematical Statistics*, 19(3):326–339, 1948. 50