ScrewBox: a Randomized Certifying Graph-*Non*-Isomorphism Algorithm

Martin Kutz[*] Pascal Schweitzer[*]

Abstract

We present a novel randomized approach to the graph isomorphism problem. Our algorithm aims at solving difficult instances by producing randomized certificates for *non*-isomorphism. We compare our implementation to the de facto standard nauty. On many of the hardest known instances, the incidence graphs of finite projective planes, our program is considerably faster than nauty. However, it is inherent to our approach that it performs better on pairs of non-isomorphic graphs than on isomorphic instances.

Our algorithm randomly samples substructures in the given graphs in order to detect dissimilarities between them. The choice of the sought-after structures as well as the tuning of the search process is dynamically adapted during the sampling. Eventually, a randomized certificate is produced by which the user can verify the non-isomorphism of the input graphs. As a byproduct of our approach, we introduce a new concept of regularity for graphs which is meant to capture the computational hardness of isomorphism problems on graphs.

1 Introduction

The computational complexity of graph isomorphism remains unresolved for over thirty years now. No polynomial-time algorithm deciding whether two given graphs are isomorphic is known; neither could this problem be shown to be NP-complete. Graph isomorphism is one of the two remaining open problems from Garey and Johnson's famous list [5] of computational problems with this unsettled complexity status.

For several special classes of graphs, polynomial-time algorithms are known, the most prominent being planar graphs [13, 6], random graphs [1], graphs with bounded eigenvalue multiplicity [2], graphs of bounded genus [4], and graphs of bounded degree [8].

In contrast to typical problems known to be NP-complete, it is not easy to devise truly difficult graph isomorphism instances. The leading graph-isomorphism solver `nauty` [9, 10] easily maps most graphs with several thousand vertices. Only highly regular graphs pose a real challenge for this program. The hardest known instances are point-line incidence graphs of finite projective planes.

In this paper, we present a new algorithmic approach to graph isomorphism. Our algorithm computes randomized certificates for *non*-isomorphism of pairs of graphs. This might come as a surprise as graph isomorphism is not even known to lie in co-NP. Based on heuristic sampling rules, we search for substructures in pairs of given graphs to find statistical evidence for non-isomorphism. As a typical application of our algorithm we see the search for new combinatorial objects like projective planes and other designs.

Our implementation, called `ScrewBox`, aims at difficult instances. Experiments on projective planes show that our program can compete with the unrivaled and widely used standard `nauty` on such graphs and even outperforms it in many cases. We present collections of graphs that are simply intractable for `nauty` but can be solved by our algorithm in several minutes.

We emphasize that our algorithm does not try to compute isomorphisms. Its performance depends on the "degree of non-isomorphism" between the given graphs. Strong similarity makes non-isomorphism verification difficult. On isomorphic pairs, our algorithm might not even terminate. There is, however, a generic way to turn an isomorphism test into an isomorphism finder. This method can also be applied to our randomized one-sided non-isomorphism test. This feature is not implemented, yet, but `ScrewBox` can already produce isomorphisms as a side-effect of the sampling process.

A central aspect of our approach is that the output provides the user with a certificate that allows them to verify a positive non-isomorphism result by running a statistical test. This is in contrast to backtracking approaches like `nauty`, which produce very simple certificates for isomorphic graphs, namely an isomorphism, but do not provide certificates for non-isomorphism.

Motivated by our sampling mechanism, we develop a new regularity notion for graphs which is meant to capture the computational hardness of isomorphism problems on graphs. This new concept of

[*] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany. `{mkutz,pascal}@mpi-inf.mpg.de`

screw regularity defines a refined hierarchy of graph properties stretching from standard (degree-)regularity to (vertex-)transitivity.

2 Background

We work with simple undirected graphs $G = (V, E)$ without self-loops. The *size* of a graph is denoted by $n = |V|$. An isomorphism between two graphs $G = (V, E)$ and $G' = (V', E')$ is a bijection $\varphi\colon V \to V'$ with $\varphi(u)\varphi(v) \in E' \Leftrightarrow uv \in E$ for all $u, v \in V$. A graph automorphism is an isomorphism from a graph to itself. The automorphism group of a graph G is the set of all automorphisms of G with composition as group operation. It is well-known that computing automorphism groups and finding isomorphisms between graphs are polynomially reducible to each other. The orbit of a vertex v in G is the set of all v-images under automorphisms of G. A standard approach to computing graph isomorphisms is to identify the orbits of a graph, which is also polynomially equivalent to graph isomorphism.

A *canonical labeling* is a function χ on all graphs that assigns labels $1, \dots, n$ to the vertices of an n-vertex graph such that $\chi_G^{-1}(i) \mapsto \chi_{G'}^{-1}(i)$ is an isomorphism for any pair (G, G') of isomorphic graphs. Computationally, this implies that as soon as we know the canonical labelings of two graphs, we can trivially check whether they are isomorphic or not. Brendan McKay's graph-isomorphism solver `nauty` uses this approach (see the `nauty` user's guide on [9]).

3 The Algorithm

A standard approach to detect whether two given graphs are non-isomorphic is via graph predicates. A *(graph) predicate* is a function on graphs that is invariant under graph isomorphisms. Simple examples of such predicates are the (multi-)set of node degrees or the set of degree sums of all neighbors of all nodes. A possibly more expressive predicate might compute the maximum flow between all pairs of vertices in a graph. If such a predicate yields different values on two graphs, they cannot be isomorphic.

On highly regular graphs, like the incidence graphs of finite projective planes, however (see Section 7 for a definition), such simple predicates are bound to fail hopelessly. On the other hand, sufficiently strong predicates appear computationally far too expensive. The idea behind our algorithm is to dynamically construct predicates that can be evaluated through statistical tests.

The algorithm tries to find certain patterns in the graph by sequentially sampling nodes from the graphs in a randomized fashion. The goal is to observe significantly different behavior of this sampling process on the two given graphs.

A single sample run draws nodes $v_1, v_2, \dots, v_n$, one after another, where each single v_t has to fulfill a certain set of rules. Such a rule determining the admissibility of a sample node v_t is called a *screw*. By tightening and loosening a screw the sampling process can be steered. Specifically, at each step t (also called a level), a set of screws determines the set A_t of admissible nodes, from which v_t is drawn at random. After that, the sampling proceeds to vertex v_{t+1}. If A_t is empty, the sample terminates and we record the depth t at which this happened. If, after running this process many times, the frequencies of these termination depths differ significantly for samples on two given graphs, we may conclude (with high probability) that the graphs are not isomorphic.

The collection of all screws for all levels $1, \dots, n$ is called the *screw box*. The construction of the screw box and the selection and tuning of the screws is a complex dynamic process that forms the core of our algorithm. Part of the construction process is the continuing evaluation of its quality. Insertions, deletions, and modifications of screws are meant to increase the statistical significance of the sampling and its speed. See Section 6 for details of our statistical analysis.

Construction of the screw box. In principle, a screw in the screw box can be an arbitrary predicate that determines whether some vertex v_t is a valid extension of the sample $v_1, \dots, v_{t-1}$ in G. In order to obtain useful screws, our algorithm begins by fixing a *pattern* $p_1, \dots, p_n$ of distinct vertices in one of the graphs, G_1. It then inserts a basic screw $S_{G_1}^{t,0}$ into each level t of the screw box. Such a screw compares the adjacencies of p_t with $p_1, \dots, p_{t-1}$ to those of a candidate v_t with the current sample $v_1, \dots, v_{t-1}$.

With this basic setup, the screw box, in principle, already achieves its full functionality. Namely, if a complete sample is found under these conditions (in G_1 or G_2), the correspondence $v_i \mapsto p_i$ is an isomorphism. This implies that, if G_1 is not isomorphic to G_2, there cannot be samples of length n in G_2 (while, of course, length-n samples will be found in G_1 with positive probability). This effect will eventually become visible in the statistics and thus prove the graphs non-isomorphic.

With highly regular graphs, this basic screw box will, however, lead to unacceptably long running times. The point of our approach is to install more powerful screws that guide the sampling process much more effectively. Such higher-level screws do not only consider the correctness of the sample itself but take its relative position within the whole graph into account. As a result, the average sample length increases. This should, however, be seen only as a side effect since our main goal is to increase statistical significance.

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 screws has to be done with great care. A main feature of our algorithm is the self-adaptive behavior of the screw-box's construction process. Screws are not placed by the user but they are tested for effectiveness during the sampling. On an easy instance, for example, no expensive screws will get installed, whereas a highly regular graph will induce very few expensive screws at crucial positions of the pattern. There is no need for the user to specify in advance the difficulty or special properties of the input graphs.

We want to emphasize that sampling for a pattern does not mean that we are implicitly computing subgraph isomorphism. The main goal of the screw box is *not* to find very long samples but to create significant deviation in the termination levels on the two graphs.

4 The Randomized Certification Model

The typical graph-isomorphism algorithm takes two graphs and either returns an isomorphism between them or simply returns "non-isomorphic." While in the former situation, the user can verify the correctness of the output by simply checking the isomorphism, in the latter, he is bound to trust the algorithm, which might not be very satisfying. This is typical for an NP-problem not known to be in co-NP: A positive answer is easy to verify, while the correctness of a negative outcome can usually only be recognized through verification of the algorithm itself.

Our algorithm creates the exactly opposite situation. When it terminates, it has found with high probability a screw box that can be used to establish a difference between the 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. He only needs to convince himself that sampling with the screw box is invariant under graph isomorphism. For the screws we use, this is quite an obvious property. The user employs the screw box to repeat the statistical test from the construction phase with his desired error probability to confirm the non-isomorphism claim.

Note that the algorithm will almost surely terminate. It either outputs a screw box that separates two non-isomorphic graphs or will produce an isomorphism. In any case it will produce a certified answer. Of course, this two-sided termination guaranty is a theoretical concept. The aim of the screw-box approach is to identify non-isomorphism. (There are general ways to turn a one-sided non-isomorphism test into an isomorphism finder with $O(n^2)$ overhead, but we cannot go into detail here.)

We should mention that we do not claim that our algorithm shows graph isomorphism to lie in co-NP. We do not expect our algorithm to have polynomial running time. Constructing such an algorithm—if possible—seems to require much deeper structural insight. We shall come back to this issue later in Section 7.

5 Screws

Let S be a function on pairs $(G, \bar{v})$, where G is a graph and $\bar{v} = (v_1, \ldots, v_t)$ is a sequence of vertices of G. Such an S is called a *screw* if it is invariant under graph isomorphism, i.e., for any isomorphism $\varphi\colon G \to G'$ we have $S(G, \bar{v}) = S(G', \varphi(\bar{v}))$ for all $\bar{v}$. The sequence $\bar{v}$ is called the *sample* as it corresponds to the vertex sequences that are generated during the sampling process of our algorithm.

The simplest screws used by our algorithm consider only adjacencies among the vertices within the sample. It completely encodes how the last sample vertex v_t is connected to the other sample vertices. For a formal definition we need a "characteristic function" λ of E. For any pair (u, v) of distinct vertices we define $\lambda(u, v) = 1$ if $\{u, v\} \in E$ and $\lambda(u, v) = 0$ otherwise. On the diagonal we let $\lambda(u, u) = -1$. The basic screw $S^{t,0}$ is then defined by

$$(5.1) \qquad S^{t,0}(G, \bar{v}) = \big(\lambda(v_1, v_t), \ldots, \lambda(v_{t-1}, v_t)\big).$$

Consider the pattern $p_1, \ldots, p_n$ in the graph G_1 which the algorithm fixes in the beginning. During the sampling process on graph G_i, when the next vertex v_t is to be selected, we compare the screw value of the sample to that of the pattern. The candidate v_t is admissible (w.r.t. $S^{t,0}$) if

$$S^{t,0}\big(G_i, (v_1, \ldots, v_t)\big) = S^{t,0}\big(G_1, (p_1, \ldots, p_t)\big).$$

Using this basic screw as a building block, we construct stronger ones, $S^{t,1}$, by taking the whole graph into account, and not just adjacencies within the sample. We look at all possible ways of extending the sample (including the candidate v_t) by one further node $u \in G$. For each u we compute all adjacencies between $v_1, \ldots, v_t$ and u. The resulting values, for all u, are collected in a multiset. Additionally, we add the adjacencies of the candidate v_t with the rest of the sample. Formally, we define

$$\begin{aligned} S^{t,1}(G, \bar{v}) \quad &= \quad \big\{S^{t+1,0}\big(G, (v_1, \ldots, v_t, u)\big) \bigm| u \in G\big\} \\ &\quad \cup \ \big\{S^{t,0}\big(G, (v_1, \ldots, v_t)\big)\big\}, \end{aligned}$$

This construction continues recursively. Again, we have a sample $\bar{v}$ ending on candidate v_t. For each further node $u \in G$, consider all nodes $w \in G$ and evaluate $S^{t+2,0}\big(G,(v_1,\ldots,v_t,u,w)\big)$. Formally, for all $k > 0$, we define,

$$\begin{aligned} S^{t,k}(G,\bar{v}) \;&=\; \big\{S^{t+1,k-1}\big(G,(v_1,\ldots,v_t,u)\big) \;\big|\; u \in G\big\} \\ &\;\cup\; \big\{S^{t,0}\big(G,(v_1,\ldots,v_t)\big)\big\}. \end{aligned}$$

We call such an $S^{t,k}$-screw a *k-level screw* (on a sample of length t). For completeness, we have to define the trivial screw $S^{0,0} \equiv ()$.

At this point the attentive reader might want to meditate over the fact that a k-level screw does *not* simply list all k-tuples of G-vertices and record their adjacencies with the sample. Our screws are much stronger because they filter the adjacency information of possible sample extensions hierarchically. A more detailed discussion of the mechanics of our screws would be beyond the scope of this exposition.

The k-level screws $S^{0,k}$ (on empty samples) form a hierarchy of graph-isomorphism invariants, where on the highest level $k = n$ we have $S^{0,n}(G,()) = S^{0,n}(G',())$ if and only if G and G' are isomorphic. In other words, $S^{0,n}$ is a function that classifies the isomorphism types of n-vertex graphs.

Computational Cost. The evaluation of a k-level screw takes $O(t \cdot n^k)$ time since it entails k nested loops over the vertices of the graph. In practice, we do not compute multisets of multisets, of course, but instead hash them to single integers.

Naively evaluating screws repeatedly the way they are defined would soon become impractical, already for 2-level screws. In our implementation, we perform a lot of optimization to strip the screws of irrelevant node types (many nodes turn out to have no effect on the value of a screw) and of superfluous adjacency tests. Eventually, we work with highly fine-tuned screws that have very good separation properties at low cost. This screw tuning is an integral and complicated part of our algorithm and is indispensable for achieving acceptable running times.

6 Stochastics

The data we collect during the sampling process consists of the termination levels of the samples. A *histogram* is a map $H\colon [n] \times [2] \to \mathbb{N}$. The empty histogram is the all-zero function. A single sample run on graph G_i terminating on level t corresponds to the histogram with $H(t,i) = 1$ and 0 elsewhere. The histogram of a set of sample runs is the sum of the histograms of all samples in the set. In particular, extending the sample set by one increases exactly one entry in the histogram.

If after equally many samples on both graphs, G_1 and G_2, the values $H(t,1)$ and $H(t,2)$ differ significantly for some $t \le n$, we have discovered evidence that the graphs are not isomorphic. However, the deviation on individual levels of the histogram is usually not significant enough for us to be able to claim non-isomorphism of the graphs with sufficient confidence. Therefore, we filter our histograms. A *filter* is a function F that maps a histogram to a pair (a,b) of integers in the following way. Each filter is specified by coefficients $\sigma_1,\ldots,\sigma_n \in \{-1,0,+1\}$ and evaluates as

$$\begin{aligned} F(H) \;&=\; \sum_{t\,:\,\sigma_t\,=\,+1} \big(H(t,1),H(t,2)\big) \\ &\quad+ \sum_{t\,:\,\sigma_t\,=\,-1} \big(H(t,2),H(t,1)\big). \end{aligned}$$

In words, the coefficients σ_t specify how a histogram level contributes to $F(H)$: by direct addition, by swapped addition, or not at all. Hence, $a + b$ is at most $2 \cdot N$, if N sample runs have been performed on each graph. A good filter will choose the coefficients in such a way that the sampling process produces two integers with significant imbalance. Setting the coefficient of a level to 0 allows to exhibit significant difference within seldomly occurring levels even if they are among frequent levels with almost no imbalance.

In case the given graphs are isomorphic, of course, the expected values of a and b coincide—irrespective of the filter we apply. Given a significance level α, the algorithm can provide a test number N and values for a and b for which one can conclude the graphs to be non-isomorphic with probability $1 - \alpha$.

During the construction of the screw box, various test filters are applied in order to estimate its quality. Once we are sufficiently convinced of the suitability of the screw box, we freeze the screw box and run further samples in order to determine a promising filter.

7 Highly Regular Graphs

When dealing with the graph isomorphism problem it is natural to ask how it is possible to measure the hardness of a graph. A graph is hard to understand if it has vertices that appear the same but do not lie in the same orbit. We introduce a new notion of regularity that aims to capture the hardness for an algorithm to understand its structure and differentiate it from other graphs that are almost isomorphic.

A graph is said to be *regular* if all its vertices have the same degree, and it is *strongly regular* if in addition there exist two numbers μ and ν such that every adjacent pair of vertices has exactly μ common neighbors and every non-adjacent pair of vertices has exactly ν common neighbors. It seems, however, that these regularity notions are not strong enough to explain which graphs are difficult in isomorphism testing.

Projective planes. In contrast to typical NP-complete problems, it appears very difficult to devise hard instances for the graph isomorphism problem. It is generally accepted that the incidence graphs of finite projective planes confront graph isomorphism algorithms with great challenges.

A (finite) *projective plane* is a bipartite graph satisfying the following conditions:

- each pair of vertices in one partition class has exactly one common neighbor,
- there exist at least four vertices in one partition class, no three of them with a common neighbor.

The vertices in one bipartition class are called *points* and those in the other *lines*. A line ℓ and a point p are called *incident* if $\{\ell, p\}$ is an edge. It is an easy but insightful exercise to show that projective planes are regular. The *order* of a projective plane is defined to be $\eta = d - 1$, where d denotes the (uniform) vertex degree. It is easily verified that a projective plane of order η contains exactly $\eta^2 + \eta + 1$ points and the same number of lines.

Projective planes can be constructed as the containment graphs of the 1 and 2-dimensional subspaces in 3-dimensional vector spaces over finite fields. The order of such a plane is, by construction, the size of the underlying field. Projective planes arising this way are called *algebraic*. There exist non-algebraic projective planes, all known ones being of prime-power order, too. It is a famous open question whether there exist projective planes of non-prime-power order.

***k*-Screw regularity.** The concept of strongly regular graphs does not seem to capture difficulty for graph isomorphism, especially since projective planes do not have this property.

We propose new regularity conditions that aim to parameterize the computational hardness of a graph in the context of isomorphism testing. The concept is based on our k-level screws, which were designed to detect irregularities in graphs.

Definition 7.1. *A graph G is called k-*screw-regular *if for any two vertices u, v of G we have*

$$S^{1,k}(G,u) = S^{1,k}(G,v).$$

It is easy to see that 1-screw-regular graphs are exactly the regular graphs. Any strongly regular graph is also 2-screw-regular. In fact, strongly regular graphs can be described exactly as those regular graphs for which $S^{2,0}\big(G,(u,v)\big) = S^{2,0}\big(G,(u',v')\big)$ implies $S^{2,1}\big(G,(u,v)\big) = S^{2,1}\big(G,(u',v')\big)$ for all u, v, u', v' in G. For further aspects of strongly regular graphs we refer to the survey [3].

Observe that the $S^{1,n-1}$-screws characterize orbits: Two vertices u, v lie in a common orbit if and only if $S^{1,n-1}(G,u) = S^{1,n-1}(G,v)$. Hence, the n-screw-regular graphs are exactly the transitive graphs. This goes well with intuition because transitivity is a very strong regularity condition: all vertices appear the same.

Regularity of colored graphs. A *colored graph* is a graph $G = (V,E)$ together with a coloring function $\chi\colon V \to \mathbb{N}$. For colored graphs, we restrict our regularity notions to individual color classes. That is, for example, a regular colored graph is a graph in which all vertices of the same color have the same degree. (Formally, the characteristic function λ in Equation 5.1 now also has to respect colors.)

A colored graph G is called *k-screw-regular* if for any two vertices u, v of G that have the same color, i.e., $\chi(u) = \chi(v)$, we have $S^{1,k}(G,u) = S^{1,k}(G,v)$. By using colors, we are able to restrict the concept of screw regularity to parts of a graph. This extends its applicability to a larger class of graphs. For example, an orbit coloring makes any graph k-screw regular for arbitrary k.

Theorem 7.1. *If there exists a fixed k_0 s.th. whenever a colored graph is k_0-screw-regular it is also k-screw-regular for all $k > k_0$, then graph isomorphism is in P. Precisely, graph isomorphism can then be solved in $O(n^{k_0+2})$ time.*

This leads us to the following fundamental problem: *Do there exist arbitrarily large k such that there are (colored) graphs that are k-screw-regular but not $(k+1)$-screw-regular?* We have verified that all projective planes are 7-screw regular but there exist examples that are not 8-screw regular. Maybe large projective planes can yield an affirmative answer to the above question.

8 Computational Results

We implemented our algorithm in `C++`, without the use of special graph or matrix libraries, representing graphs as simple adjacency matrices. We gave our program the name `ScrewBox`.

We performed many tests on a 2.4 GHz AMD Opteron machine with one 1 GB RAM, running Linux. As a benchmark, we ran all test instances through `nauty` as well. Our test runs deal with projective planes. These are the hardest known instances for `nauty` and as it turned out, they also confront `ScrewBox` with the greatest challenges. On easier graphs our code

		proj-16		unions						
		alg	n'alg	1	2	3	4	6	8	10
nauty	avg.	0 s	2 m	3 m	79 m	368 m	441 m	1101 m	2096 m	–
ScrewBox	50 %	2 s	2 m	1 m	2 m	4 m	7 m	13 m	19 m	32 m
	95 %	4 s	37 m	31 m	71 m	129 m	338 m	479 m	843 m	1403 m

		proj-27			joins						
		alg	n'alg	flag	1	2	3	4	6	8	10
nauty	avg.	4 s	421 m	64 h	1716 m	–	–	–	–	–	–
ScrewBox	50 %	18 s	39 m	73 h	1 m	2 m	4 m	6 m	13 m	25 m	34 m
	95 %	39 s	167 m	–	24 m	79 m	148 m	226 m	595 m	936 m	1277 m

Figure 1: Running times for ScrewBox and nauty (dashes indicating that computations did not finish within three days).

is by orders of magnitudes faster than on projective planes but still drastically slower than nauty. Yet, we emphasize that ScrewBox is not specially tuned for projective planes, although we see possibilities to do so.

Direct comparison of ScrewBox to nauty is somewhat difficult because nauty detects isomorphism (and also non-isomorphism, of course) via canonical labelings of single graphs, while our algorithm always works on pairs of graphs. We only tested non-isomorphic pairs of graphs. The algorithm has been set to ensure an error probability of at most 0.05%.

Figure 1 gives an overview of the results of our test. Since the deterministic running times of nauty turned out to vary only slightly within the considered graph classes, we simply list their averages. For ScrewBox, we performed many runs on distinct pairs of graphs within the respective class. We observed a large deviation between the running times even on the same pair of graphs. Therefore, we list the time it took 50% respectively 95% of the runs to complete (successfully).

Projective planes. We used all known projective planes of order $2^4 = 16$ and $3^3 = 27$, which we took from Eric Moorhouse's and Gordon Royle's web pages [11, 12]. There are 13 known planes of order 16 and 8 of order 27. (As geometric structures of points and lines, there are actually 22 and 13 known planes of these orders, respectively, but considered as incidence graphs, planes cannot be distinguished from their duals.) Apart from the algebraic ones, we did not have access to any planes of higher order. 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. It turned out that the difficulty of the planes varies. For both, nauty and ScrewBox, algebraic planes ("alg") are much easier to solve than the non-algebraic ones ("n'alg"). Therefore we separated all computations that involved algebraic planes from the rest. Two exceptionally difficult planes of order 27, called "flag4" and "flag6" on [11], are also listed separately. To solve that pair, 50% of the SB runs took slightly longer than nauty needed for a labeling of one graph. Be aware, though, that nauty has to label both graphs in order to detect non-isomorphism.

Unions and joins. Unfortunately, we could not find any non-algebraic projective planes of order 32 and above on the web. In order to devise larger and more difficult instances, we combined 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 and $G * H$ denotes the *join* of graphs G and H, i.e., the disjoint union of G and H together with all edges joining G and H.

We ran ScrewBox and nauty on the unions $1 \cdot P, \dots 10 \cdot P$ and the joins $1 \cdot P * F, \dots, 10 \cdot P * F$ for four non-algebraic projective planes P of order 16. Here F denotes the *fano plane*, the unique projective plane of order 2. The right sections in Figure 1 show the running times of ScrewBox and nauty on these graphs.

ScrewBox proved very robust under the above graph

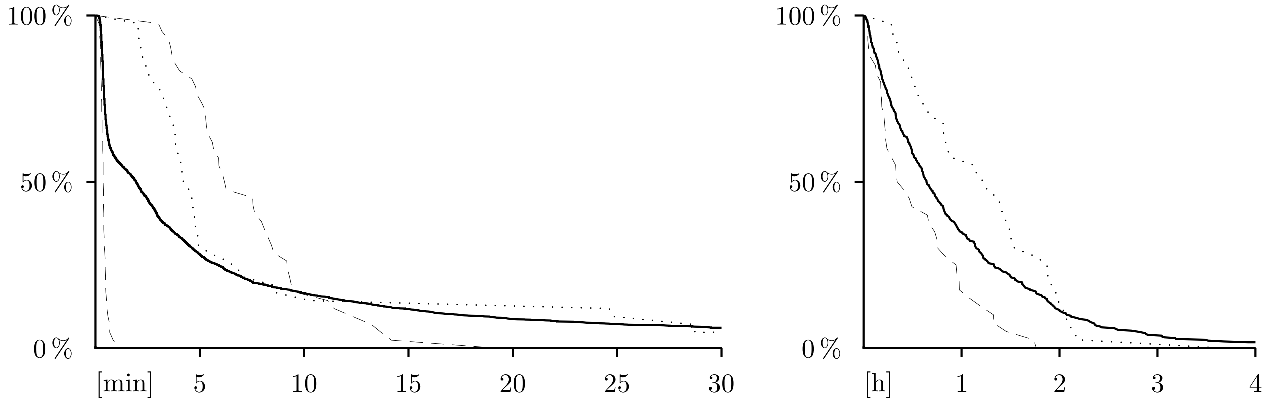

Figure 2: Termination times of our algorithm on projective planes of order 16 (left) and 27 (right); shown as portion of non-completed tasks after given time (thick line: all pairs; dashed and dotted lines: selected pairs).

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 obviously not lead to an explosion of running times. In particular, joining an extra fano plane to the disjoint unions does not create any problems for our code. These observations match well with our understanding of the sampling approach. The sampling tends to invest most of its resources in the "interesting" regions of the graph. The fano plane in the above examples does thus not interfere with the discrimination of the base graphs P.

It turns out that for `nauty`, already the smallest instances of this collection are prohibitively difficult. The large disjoint unions take several hours to compute. For one of the planes, the 9-fold union did not finish within a week. Joining the fano plane to the unions had 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 principle, this does not come as surprise. In order to obtain canonical labelings, `nauty` has to establish isomorphisms between all components. The extra fano plane seems to complicate this task.

We remark that `nauty` offers a number of configurations to adapt it to different classes of graphs. On each instance, we tried `nauty` with and without the "cellfano2" option, which is recommended for computation of projective planes. The table only considers the faster run for each graph. It should be said that it might be possible that experts could further tune `nauty` for the specific types of graphs we considered. But this is true for `ScrewBox`, too.

Variation of running times. Figure 2 shows the distribution of running times for `ScrewBox` on projective planes of orders 16 and 27 in more detail. The curves depict the portions of completed tasks after a given time. The thick lines represent the 21 runs on all pairs (except those involving the algebraic planes) and the dotted and dashed lines show the behavior of `ScrewBox` on a few selected pairs of graphs, indicating the variation of difficulty amongst different instances. Future work on `ScrewBox` will include the integration of mechanisms that decrease the variation by avoiding exceptionally long runs.

9 Technical Details

When approaching the isomorphism problem, it is reasonable to extract obvious information one can gain from a graph in a preprocessing step. One such step could be to color vertices that are already known not to lie in the same orbit differently. For example, nodes with different degrees or nodes with different neighborhoods should be separated this way. A more sophisticated step is to mark orbits for pairs of vertices. We call a matrix which has one entry for every ordered pair of vertices such that the entries are invariant under graph isomorphism a *pairlabel matrix.* It corresponds to an edge coloring of the complete graph on n vertices.

The adjacency matrix of a graph itself is such a pairlabel matrix. To richen the information in this matrix, our algorithm applies several deterministic manipulation steps to it. Repeated squaring is one of these steps, it is the most expensive and powerful one. It implicitly encodes distance information into the matrix. Specifically, the algorithm interprets the adjacency matrix as edge colored graph, colors being integers, and repeatedly tries to refine the coloring, by differentiating nodes by colors of incident edges and by adding the square of the current pairlabel matrix to itself. In a projective plane the only gain from this procedure is that pairs of vertices from the same partition class are colored different than pairs of vertices in different partition class, i.e that consist of a point and a line.

Since our algorithm aims at very difficult graph instances, the time consumption of this extra computation is negligible. The algorithm could dispense with this preprocessing, but that would result in a significant increase of running time during the sampling process since each sample has to detect otherwise obvious differences among the nodes every single time from scratch. The additional information is also used to tune the screws. When a k-level screw rejects a candidate there is a difference in the evaluation of the screw on the current sample opposed to it's evaluation on the pattern. The algorithm determines what adjacencies really have to be evaluated to recognize a difference. A new screw will be used from now on, one that only calculates this necessary part. For example a 2-level screw could be restricted to only consider neighbors of the canditate with a certain color and their incident edges of another certain color.

The choice of the pattern has turned out to play a significant role. Since there is no need to generate the whole pattern in advance, it is produced on demand. Whenever the length of a sample exceeds all lengths of previous samples, the pattern will be extended. There are different generic ways of how to generate it. For example, simply randomly pick a vertex, or differentiate the nodes into classes and picking such a class uniformly at random. What has turned out to be a very useful generation method is to select a singleton class whenever possible and to pick a node in the largest class otherwise. This way the sampling collects obvious information very cheaply and can determine erring choices faster.

10 Conclusion and Outlook

Our approach of solving the graph isomorphism problem by randomized search for non-isomorphism predicates has proven competitive with the current standard and is even able to outperform it on large difficult instances, without being specially tuned for them.

Our implementation, `ScrewBox`, still offers vast room for improvement. Apart from low-level optimization, significant speed-up should be possible through the integration of some natural extensions. These include k-level screws for $k > 2$, randomized screws, dynamic editing of the pattern (which is currently fixed once generated) and parallel evaluation of several patterns. User-specified patterns could allow to tune the algorithm towards special classes of graphs.

By building screw boxes in advance, our approach offers the possibility to prepare a library of graphs for non-isomorphism tests against a large collection of unclassified graphs. This might be a very effective strategy in computer-assisted searches for new combinatorial objects—like non-algebraic projective planes of large orders.

One of the most important extension of `ScrewBox` will certainly be the development of an isomorphism finder. There exist two natural ways to achieve this goal. The more standard solution would use our non-isomorphism certificates to identify orbits. Another approach would be to devise stronger screws to directly steer the sampling process towards an isomorphism. At the current state the algorithm already finds isomorphisms this way, for all easy graphs and even for most projective planes of orders 16 and some of order 27.

References

[1] Laszlo Babai, Paul Erdös, and Stanley M. Selkow. Random graph isomorphism. *SIAM Journal on Computing*, 9:628–635, 1980.

[2] László Babai, D. Yu. Grigoryev, and David M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proceedings of STOC*, pages 310–324, 1982.

[3] Peter J. Cameron. *Topics in Algebraic Graph Theory*, chapter Strongly regular graphs. Cambridge Univ. Press, 2004.

[4] I. S. Filotti and Jack N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *Proceedings of STOC*, pages 236–243, 1980.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, New York, 1979.

[6] John E. Hopcroft and J. K. Wong. Linear time algorithm for isomorphism of planar graphs. In *Proceedings of STOC*, pages 310–324, 1974.

[7] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem.* Birkhäuser, 1993.

[8] 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.

[9] B. D. McKay. The nauty page. `http://cs.anu.edu.au/∼bdm/nauty`.

[10] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[11] G. Eric Moorhouse. Projective planes of order 27. `http://www.uwyo.edu/moorhouse/pub/planes27`.

[12] Gordon Royle. Projective planes of order 16. `http://www.csse.uwa.edu.au/∼gordon/remote/planes16`.

[13] Robert Endre Tarjan. A V^2 algorithm for determining isomorphism of planar graphs. *Information Processing Letters*, 1(1):32–34, 1971.