Attacks on Hard Instances of Graph Isomorphism

Greg Tener and Narsingh Deo
School of Electrical Engineering and Computer Science
University of Central Florida, 32816-2362
{gtener, deo}@cs.ucf.edu

Abstract

The Graph Isomorphism (GI) problem asks if two graphs are isomorphic. Algorithms which solve GI have applications in but not limited to, SAT solver engines, isomorph-free generation, combinatorial analysis, and analyzing chemical structures. However, no algorithm has been found which solves GI in polynomial time, implying that hard instances should exist. One of the most popular algorithms, implemented in the software package **nauty**, canonically labels a graph and outputs generators for its automorphism group. In this paper we present some methods that improve its performance on graphs that are known to pose difficulty.

1 Introduction

A graph $G = (V, E)$ models a relation E over the vertex set V. Two graphs G and H are isomorphic if they both represent the same relation, but with possibly different names attributed to the vertices. Graphs are a powerful model and can be used to represent any combinatorial object [2], therefore to solve the isomorphism problem for an arbitrary combinatorial object, it suffices to solve the problem for graphs.

The graph isomorphism problem is interesting for several reasons. One reason is that despite the significant amount of effort directed towards discovering a polynomial-time algorithm, none has been found. This effort has however, lead to polynomial-time algorithms for several classes of graphs including but not limited to those with bounded degree [11], bounded eigenvalue multiplicity [4], and

bounded genus [16]. This implies that some instances should exist which take an inordinate amount of time for current general case algorithms. On the other hand, graph isomorphism has not been shown to be NP-Complete. One reason to suspect that testing isomorphism is not NP-Complete is that its counting problem (determining the number of isomorphisms between two graphs) is polynomial-time equivalent to determining if they are isomorphic. No NP-Complete problem is known to be polynomial-time equivalent to its counting problem [12]. Also, if graph isomorphism is NP-Complete, then the polynomial hierarchy would collapse, which is unlikely [9].

Aside from counting the number of isomorphisms, the graph isomorphism problem is polynomial-time equivalent to finding all of the symmetries in a graph [12]. The set of all permutations of the vertices that fix the edge set is called the automorphism group and is denoted by Aut(G). The automorphism group has many practical uses. Chemists use it to determine the symmetry in molecules [20]. In combinatorics it is a fundamental tool used in the analysis and construction of new combinatorial objects [22]. It is also used to efficiently and exhaustively generate isomorph-free objects such as graphs [15] or relational models [8]. In computer science, state of the art SAT solvers take advantage of symmetry in formula to achieve exponential speedup on many real-world instances. See [6] and [1] for examples. This is useful for circuit verification since human design artifacts often introduce symmetry.

Another similar problem to graph isomorphism is finding a canonical labeling of a graph [5]. As an example of a canonical labeling consider all possible adjacency matrices of a graph (there are $\frac{n!}{|\mathrm{Aut}(G)|}$ of them). Sorted lexicographically, row by row, the minimum (or maximum) element is an example of a canonical labeling. Two graphs are isomorphic (denoted with $\cong$) if and only if their canonical labellings are identical. If C is a function that returns a canonical labeling of a graph, then $G \cong H \Leftrightarrow C(G) = C(H)$.

Solving the canonical labeling problem implies a solution to graph isomorphism. However it is still unknown if knowledge of Aut(G) can be used to find a canonical labeling of a graph in polynomial time. For a discussion on the relationship between graph isomorphism and canonical labeling, see [5]. Another method of determining isomorphism uses Aut($G \cup H$), where $G \cup H$ is the disjoint union of G and H. The graphs G and H are isomorphic if and only if Aut($G \cup H$) contains an automorphism that swaps the vertices of G with those of H. The fastest isomorphism algorithms compute a canonical labeling and output generators for the automorphism group which are also used to prune the search for a canonical labeling. One of the fastest and most widely used algorithms is implemented in a software package called **nauty**written by Brendan McKay [14].

In this paper we describe the inner workings of **nauty**with particular attention for the degrees of freedom allowed in implementing the main components. We show that the target cell choice can drastically affect the algorithm's runtime on hard instances.

2 Preliminary Concepts

In order to describe the algorithm some connections need to be drawn between graphs, partitions, and permutations.

2.1 Definitions and Notation

A graph $G = (V, E)$ consists of a finite vertex set V and edge set $E \subseteq V \times V$. Under this definition the edges need not be symmetric and the vertex set is assumed to be $\{1, 2, \ldots, n\}$ unless otherwise specified. The action of a permutation $\delta \in S_n$ on a vertex $u \in V$ is denoted v^δ where $v^\delta = \delta(v)$. The action a permutation δ induces on a graph is

$$\begin{aligned} G^\delta &= (V^\delta, E^\delta) \\ &= \left(V, \left\{(u^\delta, v^\delta) \ : \ (u, v) \in E\right\}\right) \end{aligned}$$

from which $\mathrm{Aut}(G) = \{\gamma \in S_n \ : \ G^\gamma = G\}$. We typically use δ for arbitrary permutations while γ is used for automorphisms. With this notation a canonical labeling C has the property that $C(G^\delta) = C(G)$ for all $\delta \in S_n$.

2.1.1 Partitions and Permutations

The automorphism group naturally partitions the vertices into equivalence classes called *orbits*. Two vertices u and v are in the same orbit if an automorphism $\gamma \in \mathrm{Aut}(G)$ exist such that $u^\gamma = v$. This is denoted by $u \sim v$ if the automorphism group is clear. For example, using the disjoint cycle notation for permutations, if $H = \{(1\ 2), (3\ 4)\}$ then the orbits of $\langle H \rangle$, the group generated by H, are $\{1, 2\}$ and $\{3, 4\}$. Viewed as an ordered partition (called just partition from now on) and sorted by the minimum element in each orbit, the orbits are [1 2 | 3 4]. This partition is composed of two *cells* of size two each of which is a set of integers. If instead $H = \{(1\ 2), (3\ 4), (2\ 3)\}$ then $\langle H \rangle$ has only one orbit, namely $\{1, 2, 3, 4\}$ or as a partition [1 2 3 4]. The partition p = [1 2 | 3 4] is *finer* than q = [1 2 3 4] denoted $p \preceq q$ because p can be created by splitting the cells of q. Under this definition a partition is finer than itself. Extending this terminology to groups, a group H is finer than a partition π if each orbit of H is a subset of some cell of π. This applies to permutations $\delta \in S_n$ as well by considering the group $\langle \delta \rangle$.

For any set of permutations H the orbits of $\langle H - \{\delta\}\rangle$ where $\delta \in H$ are finer than the orbits of $\langle H\rangle$. For instance, given a set of generators H for a group, a series of successively finer partitions can be created. Using the previous example

$$\begin{aligned} \langle\{(1\ 2),(3\ 4),(2\ 3)\}\rangle &\to [\ 1\ 2\ 3\ 4\ ] \\ \langle\{(1\ 2),(3\ 4)\}\rangle &\to [\ 1\ 2\ |\ 3\ 4\ ] \\ \langle\{(1\ 2)\}\rangle &\to [\ 1\ 2\ |\ 3\ |\ 4\ ] \\ \langle\{\}\rangle &\to [\ 1\ |\ 2\ |\ 3\ |\ 4\ ]. \end{aligned}$$

A sequence of successively finer partitions is called a *partition nest*. The indexing of the levels in a nest begins at 0, so here the partition on the 0th level has only one cell. Any partition consisting of a single cell is called the *unit* partition. At the final (3rd) level is a *discrete* partition [1 | 2 | 3 | 4]. A *trivial* cell has size one. A discrete partition consists only of trivial cells. Each element in a discrete partition is said to be *fixed.*

Going the other way, every partition has a set of associated permutations. A permutation $\delta \in S_n$ acts on a partition π by acting on the individual cells as in

$$\pi^\delta = [\ C_1^\delta \mid C_2^\delta \mid \dots \mid C_m^\delta\ ].$$

Then $\left\{\delta \in S_n\ :\ \pi^\delta = \pi\right\}$ is the set of all permutations finer than π. Therefore, the unit partition represents S_n and any of the $n!$ discrete partitions represent the trivial group. This provides a mechanism to specify automorphisms with restrictions on the orbits given by an initial partition π of V. The automorphism group of a graph G with respect to a partition π is

$$\text{Aut}(G,\pi) = \{\gamma \in S_n\ :\ G^\gamma = G \text{ and } \pi^\gamma = \pi\}$$

and a canonical labeling is similarly defined so that for any $\delta \in S_n$ we have that $C(G^\delta, \pi^\delta) = C(G, \pi)$.

Most of this paper assumes the context of a graph $G = (V, E)$ with possibly a partition π associated with it. If two partitions are needed the variables p and q will be used. Much of the notation is adopted from McKay's paper on **nauty**[14]. The proofs for most of the statements can be found in [14] as well.

2.2 Partition Trees

To determine $\text{Aut}(G,\pi)$ it suffices to generate each $\delta \in S_n$ and then test if $G^\delta = G$ and $\pi^\gamma = \pi$. To generate each $\delta \in S_n$ start with the unit partition, then fix each element in a non-trivial cell, and repeat until a discrete partition is reached. Figure 1 illustrates this concept for S_3.

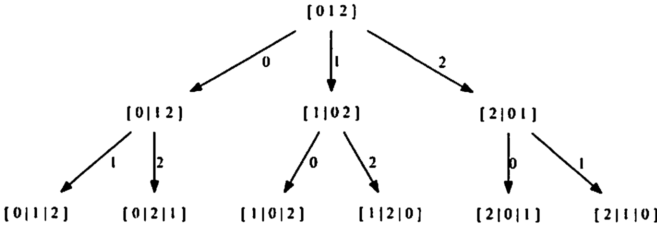

Figure 1: Partition tree for S_3. The labels on the edges indicate which element is fixed.

In this case the unit partition is the *root node* and the discrete partitions are all *terminal nodes*. The sequence of fixed elements uniquely identifies any node in the tree. This is called a node's *fixed path*. The *greatest common ancestor* of two nodes is the node identified by the greatest common prefix of the fixed paths for each node. Formally a node is a partition nest along with a fixed path. Informally and in the context of a known tree we can speak of a node as just the partition at the lowest level in the nest. As an example consider the nodes [1 | 0 | 2] and [1 | 2 | 0] with fixed paths of $(1, 0)$ and $(1, 2)$. Their greatest common ancestor is [1 | 0 2] with fixed path of just (1).

Each terminal node defines an ordering of the elements. Pick one of them to be an *identity node* which is considered to represent the identity element of the group. If p is the discrete partition of the identity node and q the discrete partition of any other terminal node then q represents the permutation $\delta \in S_n$ where $q^\delta = p$. Usually the leftmost terminal node is used as an identity node which for this tree is [0 | 1 | 2].

To generate all partitions which fix 1 start with the root partition [0 2 | 1] as seen in Figure 2 Using [0 | 2 | 1] as the identity node, $[\,2 \mid 0 \mid 1\,]^{(0\ 2)} = [\,0 \mid 2 \mid 1\,]$ so (0 2) is the only permutation in S_3 which fixes 1. This illustrates several ideas. First, the root node's partition restricts the possible permutations. Second, the order of the cells in the root partition does not affect the output, all permutations finer than the root are generated. Finally, any terminal node can serve as an identity node.

As a preview of the final algorithm, **nauty** enumerates the terminal nodes of a partition tree from left to right in a depth first manner. Any terminal nodes which induce an automorphism are stored as generators of the automorphism group. These generators are then used to prune part of the tree. Computing a canoni-

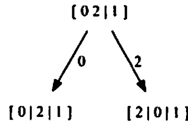

Figure 2: Partition tree for S_3 fixing 1

cal labeling then depends upon choosing an identity node that is invariant to the initial labeling of the vertices.

2.3 Refinement

Given an initial partition of the vertices π, the orbits of the group $\mathrm{Aut}(G,\pi)$ naturally partition the cells of π. In order to approximate the finer partition induced by the automorphism group, a *refinement* operation is applied to π which uses information about the graph to split the cells of π whenever possible. A refinement operation takes as input a graph $G = (V, E)$ and a partition π of the vertices V. Then, in a permutation independent way, outputs another partition which is finer than π but not finer than the orbits of $\mathrm{Aut}(G,\pi)$. The simplest example of a refinement operation does nothing at all. The ideal refinement operation refines π so that each cell is an orbit of $\mathrm{Aut}(G,\pi)$. This is somewhat unreasonable since determining the orbits of the vertices is polynomial-time equivalent to graph isomorphism [12]. In general a refinement operation is a function R with the property that for any graph G, partition π and $\delta \in S_n$

$$R(G^\delta,\pi^\delta) = R(G,\pi)^\delta \text{ and}$$
$$R(G,\pi) \geq \mathrm{Aut}(G,\pi).$$

The refinement operation used by **nauty**and many other isomorphism programs approximates the optimal. For a graph G = (V, E), let the *open neighborhood* of a vertex $u \in V$ be

$$N(u) = \{v : (u,v) \in E)\}, \text{ and for } S \subseteq V \text{ define}$$
$$\deg(u,S) = |N(u) \cap S|$$

to be the relative degree of v in S. Then, given a partition π, if two vertices u and v are in the same orbit of $\mathrm{Aut}(G,\pi)$, then their relative degree in any cell C of π

must be the same. This means that

$$u \sim v \Rightarrow \deg(u, C) = \deg(v, C), \text{ which implies that}$$
$$\deg(u, C) \neq \deg(v, C) \Rightarrow u \nsim v.$$

This provides a method of separating vertices for which $\deg(u, C) \neq \deg(v, C)$, as they must be in different orbits.

Because the refinement operation is crucial to the performance of the algorithm and for the understanding of our modifications we give a small example showing that the graph in Figure 3 has only one automorphism, the trivial one. A graph with a trivial automorphism group is called *asymmetric*.

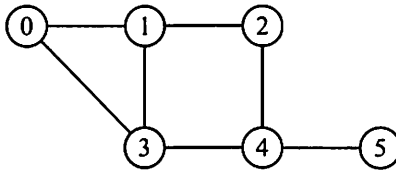

Figure 3: A graph with the trivial automorphism group

Begin with the initial partition π = [0 1 2 3 4 5], which allows any vertex to be in any other vertex's orbit. Then choose a cell C (start from the left and go to the right) called the *active cell* and sort each vertex by its relative degree in the active cell. There is only one cell in π so it is the active cell initially. The cell is split on the boundary of different degrees as shown below.

$$\begin{array}{l} [\ \mathbf{0\ 1\ 2\ 3\ 4\ 5}\ ] \longrightarrow [\ 5 \mid 0\ 2 \mid 1\ 3\ 4\ ] \\ \quad \mathit{2\ 3\ 2\ 3\ 3\ 1} \qquad\qquad \mathit{1} \quad \mathit{2\ 2} \quad \mathit{3\ 3\ 3} \end{array}$$

On the left is the partition that will be split and the active cell is in bold. On the right is the finer partition. The relative degree of each vertex with respect to the active cell is below the vertex. After the first round the vertices separate into different cells based upon their degree in the whole graph. Now choose another

active cell, and repeat the process. The final three rounds follow.

$$\begin{array}{c} [\ 5 \mid 0\ 2 \mid 1\ 3\ 4\ ] \to [\ 5 \mid 0\ 2 \mid 1\ 3 \mid 4\ ] \\ \quad 0\ 0 \quad 0\ 0\ 1 \qquad\qquad 0\ 0 \quad 1 \\ [\ 5 \mid 0\ 2 \mid 1\ 3 \mid 4\ ] \to [\ 5 \mid 2 \mid 0 \mid 1\ 3 \mid 4\ ] \\ \quad 2\ 1 \quad 1\ 1 \qquad\qquad 1 \quad 2 \\ [\ 5 \mid 2 \mid 0 \mid 1\ 3 \mid 4\ ] \to [\ 5 \mid 2 \mid 0 \mid 3 \mid 1 \mid 4\ ] \\ \quad 1\ 0 \qquad\qquad 0 \quad 1 \end{array}$$

Since the refinement operation fixed each vertex, each vertex can only move to itself, meaning the only automorphism for this graph is (0)(1)(2)(3)(4)(5). Because there is only one identity node, this also provides a canonical labeling defined by the order of the vertices in the partition. Viewed as a permutation this relabeling is (5 0 2 1 4)(3). Any reference to refinement refers to the above operation unless otherwise specified.

After refinement, if two vertices u and v are in the same cell of the partition $R(G,\pi)$, then $\deg(u,C) = \deg(v,C)$ for all cells C of $R(G,\pi)$. Such a partition is called *equitable*. The refinement operation is critical to the performance and practicality of the algorithm. For almost all random graphs it refines the unit partition into a discrete partition in three rounds [3]. Even though it uses local information, it can affect the entire partition. In order to achieve efficient refinement of directed graphs, both the in and out relative degrees must be used to split cells.

2.4 Target Cell Choosing and Branching

If the refinement operation returns a discrete partition, then the automorphism group is trivial. If it does not then let π be the partition after refinement. Then π has a non-trivial cell with two vertices u, v that could be in the same orbit. Let $\pi = [\ \ldots \mid T \mid \ldots\ ]$, then define

$$\pi \circ u = [\ \ldots \mid \{u\} \mid T - \{u\} \mid \ldots\ ].$$

Before fixing vertices and branching on them, a non-trivial cell of π must be chosen. This cell is called the *target cell* and its choice vastly affects the runtime of the algorithm for difficult instances. For now a simple target cell heuristic known as *first smallest* which chooses the first smallest non-trivial cell suffices. A target cell chooser takes as input a partition π and returns a cell $\mathrm{tc}(p) = T$ of π with $|T| \geq 2$. If π is discrete then $\mathrm{tc}(\pi) = \varnothing$. Let $\pi \perp u = R(G, \pi \circ u)$ where $u \in \mathrm{tc}(\pi)$.

Then the children of a partition π are then

$$\text{children}(\pi) = \{\pi \perp u \ : \ u \in \text{tc}(\pi)\}.$$

In practice the new trivial cell $\{u\}$ is the first active cell in refinement. Branching on a vertex u means generating the child which fixes u. The terminology and concepts introduced with Figure 1 apply here. The sequence of vertices branched on is unique for each node and is called the node's fixed path.

Some freedom exists when choosing the target cell. If two partitions p and q are related by $p^\gamma = q$ where $\gamma \in \text{Aut}(G, p)$ then p and q are said to be equivalent ($p \sim q$). If no canonical labeling is needed then the only requirement on tc is that for $\gamma \in \text{Aut}(G, \pi)$,

$$p \sim q \Rightarrow \text{tc}(p)^\gamma = \text{tc}(q).$$

If however a canonical labeling is needed then for any partition π and $\delta \in S_n$

$$\text{tc}(\pi)^\delta = \text{tc}(\pi^\delta).$$

Heuristics such as first smallest, first largest, and first non-trivial are examples of target cell choosers which preserve the canonical labeling.

2.5 Automorphisms and Pruning

Here we explain how automorphisms are discovered and how they are used to prune the search.

2.5.1 Automorphisms

Now that equivalence of partitions is defined, equivalence of partition nests can be similarly defined, which leads to the method of discovering automorphisms. Define a sequence of partitions $\pi_0, \pi_1, \ldots, \pi_m$ by

$$\pi_0 = R(G, \pi)$$
$$\pi_i = \pi_{i-1} \perp v_i, \text{ for } 1 \leq i \leq m$$

where $v_i \in \text{tc}(\pi_{i-1})$. This is a partition nest with the property that $\pi_i \prec \pi_{i-1}$. A node is called terminal if its last partition is discrete. The partition at level i for $0 \leq i \leq m-1$ is π_i. The height of a partition nest is the level of the last partition. Not all terminal nodes always have the same height but we will assume they do for now. The action of a permutation $\delta \in S_n$ on a node $v = [\pi_0, \pi_1, \ldots, \pi_m]$ is

$$v^\delta = [\pi_0^\delta, \pi_1^\delta, \ldots, \pi_m^\delta]$$

Given a terminal node $\nu = [\pi_0, \pi_1, \ldots, \pi_m]$, the final discrete partition $\pi_m = [\ \nu_1 \mid \nu_2 \mid \ldots \mid \nu_n\ ]$ defines a permutation $\delta_\nu \in S_n$ where $\nu_i^{\delta_\nu} = i$. For any terminal node ν let $G(\nu) = G^{\delta_\nu}$.

Two terminal nodes ν and η are said to be equivalent, or in the same class if $G(\nu) = G(\eta)$. In this case an automorphism $\gamma \in \mathrm{Aut}(G, \pi)$ exists such that $\nu = \eta^\gamma$ and $G = G^\gamma$ because

$$
\begin{aligned}
G(\nu) &= G(\eta) \\
G^{\delta_\nu} &= G^{\delta_\eta} \\
G &= G^{\delta_\nu^{-1}\delta_\eta}
\end{aligned}
$$

and therefore $\gamma = \delta_\nu^{-1}\delta_\eta$. For any two terminal nodes ν and η the permutation $\gamma = \delta_\nu^{-1}\delta_\eta$ is said to be *induced* by ν and η. The property that $\nu = \eta^\gamma$ and $\gamma \in \mathrm{Aut}(G, \pi)$ is an equivalence relation on the terminal nodes and partitions them into equivalence classes.

The lexicographic ordering of a node's fixed path naturally orders the nodes. A node η with fixed path $(u_1, u_2, \ldots, u_m)$ is less than (is generated before) node ν with fixed path $(\nu_1, \nu_2, \ldots, \nu_m)$ if and only if ν is an ancestor of η or there exists a k such that $u_i = \nu_i$ for $1 \le i \le k-1$ and $u_k < \nu_k$. Therefore the node ζ with fixed path $(b_1, b_2, \ldots, b_m)$ where $b_i = \min(T_i)$ where T_i is the target cell at level i is the first terminal node. This corresponds to the path down to [0 | 1 | 2] in Figure 1. An *identity node* is the first terminal node generated in each equivalence class on the partition nests. As nodes are generated, the permutation $\gamma \in S_n$ they define relative to an identity node is checked and if $\gamma \in \mathrm{Aut}(G, \pi)$ then γ is stored as a generator of $\mathrm{Aut}(G, \pi)$.

2.5.2 Bases

The sequence of vertices in the fixed path $(b_1, b_2, \ldots, b_m)$ of an identity node ζ defines what is called a *base* for $\mathrm{Aut}(G, \pi)$. A base of a permutation group H is a sequence of elements (in this case $(b_1, b_2, \ldots, b_m)$) that when fixed yields the trivial group. Define a sequence of subgroups $H^{(i)}$ of H given a sequence $(b_1, b_2, \ldots, b_m)$ as

$$
\begin{aligned}
H^{(0)} &= H \\
H^{(1)} &= \left\{\gamma \in H^{(0)} \ :\ b_1^\gamma = b_1\right\} \\
&\ \vdots \\
H^{(m)} &= \left\{\gamma \in H^{(m-1)} \ :\ b_m^\gamma = b_m\right\}
\end{aligned}
$$

where $H^{(i)}$ fixes the first i points. If $(b_1, b_2, \ldots, b_m)$ is a base, then $H^{(m)} = \{()\}$, the trivial group. Here $H^{(m)}$ is called the point-wise stabilizer of $(b_1, b_2, \ldots, b_m)$.

The fixed path for any terminal node defines a base for $\text{Aut}(G, \pi)$, although not necessairly a minimal base. Let ν be a terminal node with fixed path $(u_1, u_2, \ldots, u_m)$ and $\text{Aut}(G, \pi)^{(i)}$ be the corresponding sequence of groups. Then, a terminal node η with fixed path $(v_1, v_2, \ldots, v_m)$ is equivalent to ν if and only if u_i and v_i are in the same orbit of $\text{Aut}(G, \pi)^{(i-1)}$ for $1 \leq i \leq m$.

2.5.3 Pruning

When generating the children of a non-terminal node ν of height i with target cell T_i, it could be possible to avoid branching on some vertices in T_i. Let $H^{(i)}$ be the point-wise stabilizer of ν's fixed path in $\text{Aut}(G, \pi)$. Then if it can be determined using previously discovered automorphisms that two vertices $u, v \in T_i$ where $u < v$ (u is branched on before v) are in the same orbit of $H^{(i)}$, then v can be removed from T_i because any automorphisms induced by the descendants of $\pi \perp v$ are already generated.

To prune the target cell of a node in this manner, **nauty** determines which automorphisms fix the node's fixed path and computes the orbits of the group generated by these automorphisms. Any vertex that is not the minimum element of its orbit is removed from the target cell. To be precise define $\text{fix}(\gamma) = \{v \in V : v^\gamma = v\}$ for $\gamma \in S_n$. Then for a node ν with fixed path $(u_1, u_2, \ldots, u_m)$, γ fixes ν if $\{u_1, u_2, \ldots, u_m\} \subseteq \text{fix}(\gamma)$.

When two nodes ν and η are found to be equivalent and a new automorphism discovered, the search can return to the greatest common ancestor of ν and η. Suppose an automorphism $\gamma \in \text{Aut}(G, \pi)$ is found taking terminal nodes ν to η with fixed paths $(u_1, u_2, \ldots, u_m)$ and $(v_1, v_2, \ldots, v_m)$. Let the first k elements of the base be fixed by γ. Then

$$
\begin{aligned}
\nu &= \eta^\gamma \text{ and} \\
(u_1, u_2, \ldots, u_k, \ldots, u_m) &= (v_1^\gamma, v_2^\gamma, \ldots, v_k^\gamma, v_{k+1}^\gamma \ldots, v_m^\gamma) \\
&= (u_1, u_2, \ldots, u_k, v_{k+1}^\gamma, \ldots, v_m^\gamma)
\end{aligned}
$$

and the new knowledge is gained that $u_i \sim v_i$ in $\text{Aut}(G, \pi_k)$ for $k + 1 \leq i \leq m$. Therefore the search can be resumed at level k, the level of the greatest common ancestor of ν and η.

2.6 Canonical Labellings and Indicator Functions

Using automorphisms to prune the tree reduces the number of nodes that must be generated in the same equivalence class of terminal nodes. However, using full automorphism pruning but with no refinement at least $\frac{n!}{|\mathrm{Aut}(G,\pi)|}$ identity nodes will be generated, one from each equivalence class. With refinement, the number of classes is reduced but can still be quite large. Each identity node corresponds to a possible labeling of the graph, so one must be selected as a canonical labeling.

Define an ordering of the n vertex graphs G by associating with G the n^2-bit binary number created by appending the rows of the adjacency matrix of G together. Each identity node corresponds to a labeling G^δ of G where $\delta \in S_n$. A possible canonical labeling of G is the minimum such G^δ.

The number of identity nodes can be reduced so that fewer possible canonical labellings need to be compared. If it can be determined at level i that the current node v is in a different class than the current identity node then either no children of v need to be generated, or v could become the new current identity node. A function Λ that takes as input the graph G, initial partition π, node v, and returns an element of an ordered set is called an *indicator function*. This function must be independent of the initial labeling of the graph to recover a canonical labeling.

For example, **nauty** uses the shape of the partitions in a node's nest to help distinguish nodes that are in different classes. Let shape(π) be the sequence of cell sizes of π. For the partition [0 | 1 2 | 3 4 5] this is $(1, 2, 3)$. If two nodes have a partition in their nest at the same level with different shapes, then the nodes are in different classes. Specifically,

$$\Lambda(G, \pi, v) = (\mathrm{shape}(\pi_0), \mathrm{shape}(\pi_1), \ldots, \mathrm{shape}(\pi_m))$$

with the modification that the shapes are all hashed into one value for efficiency's sake. The output inherits the lexicographic ordering of the integers, with smaller values corresponding to better nodes. Let ζ be the current identity node and $\zeta^{(i)}$ be the ancestor of ζ at level i. Then after generating a node v at level i take the following actions depending on the comparison between $\Lambda(G, \pi, v)$ and $\Lambda(G, \pi, \zeta^{(i)})$

1. $\Lambda(G, \pi, v) > \Lambda(G, \pi, \zeta^{(i)})$
   $\zeta^{(i)}$ is better, continue search at parent of v

2. $\Lambda(G, \pi, v) = \Lambda(G, \pi, \zeta^{(i)})$
   indicator values are equal, continue with search from v

3. $\Lambda(G, \pi, v) < \Lambda(G, \pi, \zeta^{(i)})$
   update ζ to be v's first terminal descendant as the new current best identity node

This ensures that at the end of the search, the identity node ζ is minimum.

At this point we can define the canonical labeling returned by **nauty**. Let Λ^* be the minimum indicator value over all terminal nodes ν. Then among all the nodes ν with indicator value Λ^*, the canonical labeling is the minimum of $G(\nu)$. We can group these concepts by defining an indicator function Λ' given another indicator function Λ to be

$$\Lambda'(G,\pi,\nu) = \begin{cases} \nu \text{ is non-terminal} & \Lambda(G,\pi,\nu) \\ \nu \text{ is terminal} & (\Lambda(G,\pi,\nu^{(m-1)}), G(\nu)) \end{cases}$$

so that the last element of the minimum identity node's indicator value is the canonical labeling.

2.7 An Example

We give a small example to illuminate many aspects of the algorithm and introduce visualizations for partition nests. Consider the disjoint union of C_3 and C_4, a 2-regular graph displayed in Figure 4. This is the smallest graph which makes

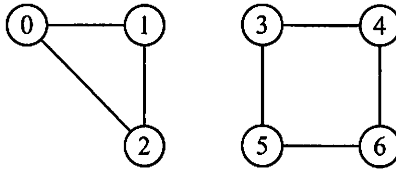

Figure 4: The disjoint union $C_3 \cup C_4$. The orbits of the vertices are $\{0, 1, 2\}$ and $\{3, 4, 5, 6\}$.

use of the indicator function. Figure 5 shows the tree generated by running the algorithm on $C_3 \cup C_4$.

The nodes are generated from left to right, with the leftmost terminal node being the first identity node. The labels of the edges signify which vertex is fixed and branched on. The fixed vertices all come from the parent's target cell. All terminal nodes except the rightmost induce a non-trivial automorphism with the leftmost identity node which allows for a jump back to the greatest common ancestor.

For an example of target cell pruning, the node [3 | 5 | 0 1 2 | 4 6] on the rightmost path to the second identity node initially has target cell $\{0, 1, 2\}$ but the orbits of the generators found so far which fix 3 are $\{0, 1, 2\}$, $\{3\}$, $\{4, 6\}$, and $\{5\}$. Because 1 and 2 are not the minimum elements of their orbits, they are removed

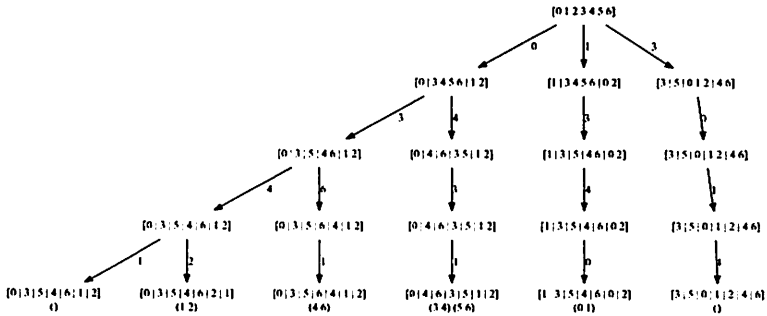

Figure 5: The full tree for $C_3 \cup C_4$.

from the target cell and only 0 is branched on.

For an example using indicator functions, the node [3 | 5 | 0 1 2 | 4 6] has a different shape than its siblings. Since the minimum indicator value is favored, and $(1,1,3,2) < (1,4,2)$ in the lexicographic ordering of the integers, the first terminal descendant of this node will be the new current identity node. At the end of the search this descendant is minimal and therefore defines the canonical labeling for $C_3 \cup C_4$.

For this example the new identity node is detected by the indicator function at the earliest possible level. This does not always happen. In other cases, the shape of each path to a terminal node could look the same for all classes, the only distinguishing factor being the final labeling of the graph. We can visualize the path to an identity node as in Figure 6. Each cell represents the possible orbits.

Figure 6: The partition nest for the first identity node of $C_3 \cup C_4$. The gray cells are the target cells. The final discrete partition is omitted.

The actual orbits can be visualized by splitting the cells given full knowledge of the automorphism group. If there are few splits, then the refinement procedure

did a good job of separating vertices which are not in the same orbit. With the full knowledge of the automorphism group, the only split in the target cells occurs at the first level, where the refinement procedure was unable to determine that $\{0, 1, 2\}$ and $\{3, 4, 5, 6\}$ were in different orbits. We call this the identity node's *finest* nest. Figure 7 illustrates the finest nest for the first identity node, with a line splitting the first level's target cell.

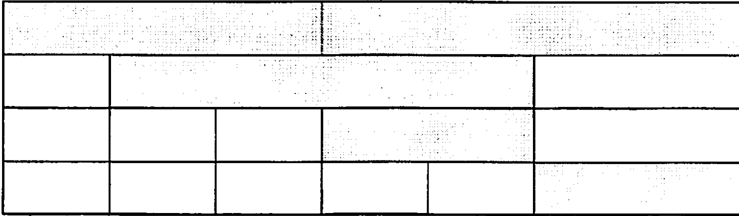

Figure 7: The finest nest for the first identity node.

Any splitting that occurs in the target cells increases the number of identity nodes. In this instance it guarantees that the tree will contain at least two identity nodes, one for each minimum element. In the general case however, let π_i be the partition at level i of the first identity node for $0 \leq i \leq m-1$ where m is the height of the nest. Then let s_i be the number of orbits of the group $\text{Aut}(G, \pi_i)$ in the target cell at level i. Then the *target cell product*, $\prod_{i=0}^{m-1} s_i$, provides a good estimate for a lower bound on the number of identity nodes that are generated. For the example of $C_3 \cup C_4$ this product is $2 \cdot 1 \cdot 1 \cdot 1$. Ideally this product is 1 in which case there is only one identity node and each terminal node induces an automorphism.

2.8 Hard Instances

There are two types of hard instances for **nauty**. The input to the algorithm could be extremely large, say a graph with several million vertices. Instances such as this arise often in the circuit verification area, and a software package using the ideas of **nauty** called **saucy** was written using sparse data structures to speed up discovering symmetries in large graphs [6]. The other type, which we focus on, involves relatively small instances which produce an inordinately large number of identity nodes. Here we describe some of these hard instances.

2.8.1 Theoretical Lower Bound

The current exponential lower bound for **nauty** was established by Miyazaki [17]. An infinite sequence of graphs and their corresponding partitions was created which drove the algorithm to generate an exponential number of identity nodes. A key component of the proof involved the choice of target cell chooser as first smallest. The order of the cells of the initial partition are chosen so that the first k target cells (all of which have size 2) get split by the full automorphism group. Then, as k increases by 1, the number of identity nodes doubles. Using the notation from [17], $\chi(Y_k)$ is a 3-regular graph with $20(k-1)$ vertices and $30(k-1)$ edges. ϑ_B is a partition of the vertices with $2k$ cells of size 2 and $2k-2$ blocks of cells of size 2, 2, and 4. The order of $\mathrm{Aut}(\chi(Y_k), \vartheta_B)$ is 2^k. Figure 8 shows the nest for the first identity node when $k = 4$.

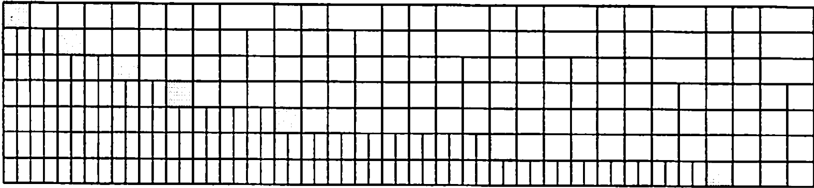

Figure 8: The partition nest for $(\chi(Y_4), \vartheta_B)$.

We can do the same analysis as for $C_3 \cup C_4$ and create a finest nest given the full automorphism group, see Figure 9. The target cell product here is $2 \cdot 2 \cdot 2 \cdot 1 \cdot 1 \cdot 1 \cdot 1$. Miyazaki's instances are constructed so that the target cell product and the number of identity nodes are equivalent. This ensures that $(\chi(Y_k), \vartheta_B)$ will force the algorithm to generate 2^{k-1} identity nodes.

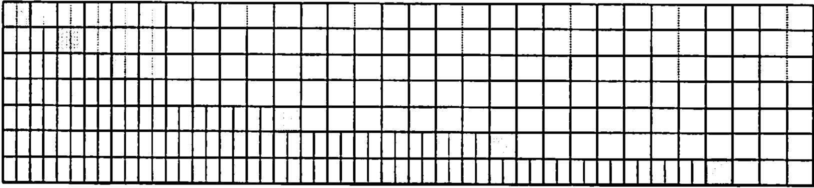

Figure 9: The finest partition nest for $(G, \pi_B)_3$.

Another set of instances with a different ordering to the cells of the initial partition, $(\chi(Y_k), \vartheta_A)$ have target cell product equal to 1 so they run in polynomial-time. The two types of graphs $(\chi(Y_k), \vartheta_B)$ and $(\chi(Y_k), \vartheta_A)$ effectively demonstrate that the target cell choice can be the difference between exponential and polynomial runtime.

2.9 Projective Planes

Aside from theoretically hard instances, some graphs derived from projective planes pose difficulty for **nauty**. For background material on projective panes see [7]. A finite projective plane of order n consists of a set of $n^2 + n + 1$ points and lines, P and L, along with a set $I \subseteq P \times L$ of incidences (p, l) describing which points lie on which lines. The set of relationships describing a projective plane can be modeled with a graph by using the set of points and lines as the vertices $(P \cup L)$ and the incidence tuples as directed edges. Undirected edges can be used as long as the points and lines lie in different cells of the initial vertex partition.

We will focus on the projective plane of order 27 with 1514 vertices and 21196 edges called flag4 found on Moorhouse's website [18]. This plane has an automorphism group of order $122472 = 2^3 \cdot 3^7 \cdot 7$ with 4 orbits which for the particular ordering of the vertices at [18] are $\{0 - 27\}$, $\{28 - 756\}$ (the points), $\{757\}$, and $\{758 - 1513\}$ (the lines). This graph is regular, so refinement does not help the initial partition.

This graph is regular, but has a relatively small automorphism group. This implies that the number of identity nodes should be large because the number of possible adjacency matrices is large. Indeed, for many hard instances, the graph is highly regular (with repsect to the usual refinement operation), but has a small automorphism group.

2.10 Hadamard Matrices

Another class of hard instances is found in graphs derived from Hadamard matrices [13]. A Hadamard matrix of order n is an $n \times n$ matrix with entries in $\{1, -1\}$. Two hadamard matrices are considered isomorphic if through some combination of column and row negations and exchanges, one matrix can be transformed into the other. A graph $G(H) = (V(H), E(H))$ is constructed so that if H_1 and H_2 are Hadamard matrices, then they are isomorphic if and only if $G(H_1)$ and $G(H_1)$ are isomorphic. To do this let

$$V(H) = \Big\{(r_i, 1), (r_i, -1), (c_j, 1), (c_j, -1) \;:\; i, j \in \{1, \ldots, n\}\Big\} \text{ and}$$
$$E(H) = \Big\{(r_i, s_r), (c_j, s_c) \;:\; s_r, s_c \in \{1, -1\} \text{ and } H_{ij} = s_r \cdot s_c\Big\}.$$

The undirected version of these graphs are regular with $4n$ vertices and $4n^2$ edges. Some Hadamard matrices of order 28 have very small automorphism groups (of size 2 for instance) with only 112 nodes and 3136 edges. While these graphs do not take hours to process, they generate large numbers of identity nodes for their small size. Because the automorphism group of size 2 and moves each vertex, after the first level, the finest partition is discrete.

3 Modifications

To experiment with different combinations of refinement operations, indicator functions, target cell choosers, and visualizations, we re-implemented the algorithm used in **nauty** with a focus on modularity (not on speed). With the standard refinement operation and indicator function, the choice of target cell affected the runtime the most.

For an extreme example, using a target cell chooser of *first smallest*, the height of the tree for the projective plane flag4 is 55 with target cell product represented as 55 digits in decimal, while using a target cell chooser of *first* yields a tree with height 5 and target cell product $21888 = 2 \cdot 12 \cdot 1 \cdot 114 \cdot 8$.

3.0.1 Greedy Target Cell Choosing

Since many difficult instances involve graphs with sparse automorphism groups and a highly regular structure, we devised a target cell chooser in an attempt to deal with these types of graphs. In the extreme case of a regular graph with only the trivial automorphism, no generators are discovered. Therefore it is intuitively desireable to reach a discrete partition as soon as possible. To do this we choose a target cell when branched upon and further refined splits the partition into the most cells. Let T be a non-trivial cell of the partition π (denoted $T \in \pi$) and $|\pi|$ be the number of cells in π. Then this target cell chooser is defined as

$$\mathrm{tc}_{greedy}(\pi) = \max_{T \in \pi} (|shape(\pi \perp \min(T))|)$$

This requires at least one refinement operation per non-trivial cell, which can be very expensive. To deal with this, we cache the target cell's index in the partition on the path down to the first identity node. Then when choosing the target cell for level i we use the cached index, and use that cell in the current partition. If a better indicator value is discovered (a different shape), the best target cells are re-computed starting at the level where a better indicator was found.

This approach of caching the target cell's index invalidates the canonical labeling because the cached value is only appropriate for terminal nodes in the same class as the first identity (or any better identity node). It is possible that for a node in a different class, the target cell choice could be different. Therefore, this approach can be used to find $\mathrm{Aut}(G, \pi)$, which is a common application.

3.0.2 Empirical Results

The greedy target cell chooser works well for flag4 and other difficult projective planes. From our observations it usually implies a smaller search tree. However, it is a greedy approach and sometimes other heuristics perform better. The target cell chooser **nauty** currently (at version 2.2) uses picks the cell which has connections with the largest number of non-trivial cells. Let C be a cell of the partition π. Define

$$\text{joins}(G, \pi, C) = \sum_{T \in \pi}^{|\pi|} N(C) \cap T > 0$$

where $N(C) = \{N(u) : u \in C\}$ and the value of $(N(C) \cap T > 0)$ is 1 if this expression is true, 0 otherwise. Then the target cell chooser used by **nauty** is

$$\text{tc}_{nauty}(G, \pi) = \max_{T \in \pi} (\text{joins}(G, \pi, T)).$$

This target cell chooser works very well on the graphs used by Miyazaki to demonstrate the exponential lower bound, much better than the greedy chooser and first smallest. However for the projective plane flag4 it creates a larger target cell product than the greedy chooser so the resultant runtime is longer.

In Table 1 we display some properties of the hard graphs for which we provide runtime statistics. The graph had.28.9 is derived from a hadamard matrix of order 28 with an automorphism group of order 2. The naming convention comes from N. J. A. Sloane's online library of hadamard matrices [19].

	$(\chi(Y_{16}), \vartheta_B)$	flag4	had.28.9
Vertices	300	1514	112
Edges	450	21196	1514
$\lvert\text{Aut}(G)\rvert$	65536	122472	2

Table 1: Properties of some hard graphs.

Our implementation, called **nishe** differs from **nauty**'s in several ways. For example, our indicator function does not hash the cell sizes, but rather performs run-length encoding. However, as Table 2 shows, our implementation processes these nodes much slower. Profiling indicates that our refinement operation is significantly slower than **nauty**'s. Both **nauty** (version 2.4) and **nishe** use sparse graph representations. In the following tables, *nauty_greedy* is **nauty** but with its target cell procedure modified to implement cached greedy choosing, and *nauty*

uses tc_{nauty}. All timings take place on an 2.2GHz Athalon X2 processor.

	nauty_greedy	nishe_greedy	nauty
Nodes	9393864	2581918	> 1255425648
Non-Generating Terminal	9264330	2408011	> 1252422563
Height	5	5 and 4	5
Target Cell Product	21888	21888	393984
Generators	10	10	> 6
Time	0.82 hrs	6.82 hrs	> 99 hrs

Table 2: Runtime statistics for flag4.

The instance for *nauty* was not run to completion. The last target cell of *nauty* had 144 splits in contrast to *nauty_greedy* which had only 8 splits, which contributes to the large runtime. The tree for *nishe_greedy* found a smaller base during the search, which became the new current identity node.

In Tables 3 and 4 the same statistics are displayed for $(\chi(Y_{16}), \vartheta_B)$ and had.28.9.

	nauty_greedy	nishe_greedy	nauty
Nodes	4016503	1543	169
Non-Generating Terminal	502528	128	2
Height	31	16	16
Target Cell Product	32768	128	2
Generators	16	16	16
Time	23.42 sec	0.31 sec	0.001 sec

Table 3: Runtime statistics for $(\chi(Y_{16}), \vartheta_B)$.

This data indicates that the target cell choice vastly affects the runtimes. The target cells are dependent upon the refinement operation, which is why many specialized invariants are implemented in **nauty**. This indirectly implies that the target cells will contain less splits if an invariant splits a cell not already split by the standard refinement operation.

	nauty_greedy	nishe_greedy	nauty
Nodes	130989	80222	130989
Non-Generating Terminal	120982	68609	120982
Height	4	3	4
Target Cell Product	471744	78624	471744
Generators	1	1	1
Time	2.89 sec	22.0 sec	2.87 sec

Table 4: Runtime statistics for had.28.9.

4 Future Work

One method of improving performance is to somehow find a better general refinement operation that can split the initial partition in regular graphs. We have experimented with a distance based refinement operation that examines neighborhoods as far away as the eccentricity of a vertex. However, this provided no benefits for any of the above three graphs.

Another method of improving performance involves dynamically choosing the best target cell based on previously discovered automorphisms. At any point in the search the number of splits in a cell can be computed. Using an idea from [10] we can choose the cell which has the fewest splits at each level. If the target cells are updated after each new automorphism is discovered, then after the full group is found the target cells will be invariant to the initial labeling and therefore the current best identity node can be used as a canonical labeling. This is also a greedy approach, but will have the advantage that it can be used for isomorphism testing efficiently.

Graphs with the trivial automorphism group would not receive any help from automorphism pruning or dynamic target cell choosing. A problem widely believed to be easier than graph isomorphism is that of determining if a graph has a non-trivial automorphism [21]. Only the refinement and indicator functions are used to process asymmetric graphs. We intend to investigate the relationship between the target cell choice and the number of nodes generated for regular asymmetric graphs and establish some lower bounds.

Finally, there might exist an infinite set of difficult instances which exhibit exponential runtime regardless of the target cell chooser. We will search for such

instances.

References

[1] Fadi A. Aloul, Arathi Ramani, Igor Markov, and Karem Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD*, 22(9):1117–1137, September 2003.

[2] L. Babai. *Handbook of combinatorics*, volume 2, chapter Automorphism groups, isomorphism, reconstruction, pages 1447–1540. MIT Press, Cambridge, MA, USA, 1995.

[3] L. Babai and L. Kučera. Canonical labeling of graphs in linear average time. *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pages 39–46, 1979.

[4] L. 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.

[5] L. Babai and E. 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.

[6] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proc. Design Automation Conference (DAC)*, pages 530–534. IEEE/ACM, June 2004.

[7] Daniel R. Hughes and Fred C. Piper. *Projective planes*. Springer-Verlag, New York, 1973. Graduate Texts in Mathematics, Vol. 6.

[8] Daniel Jackson, Somesh Jha, and Craig A. Damon. Isomorph-free model enumeration: a new method for checking relational specifications. *ACM Trans. Program. Lang. Syst.*, 20(2):302–343, 1998.

[9] Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The graph isomorphism problem: its structural complexity*. Progress in Theoretical Computer Science. Birkhäuser Boston Inc., Boston, MA, 1993.

[10] Josè Luis Lòpez-Presa and Antonio Fernàndez. Graph isomorphism testing without full automorphism group computation. *Informes Tècnicos de miembros del GSyC*, IV(3), May 2004.

[11] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. Syst. Sci.*, 25(1):42–65, 1982.

[12] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–132, 1979.

[13] Brendan D. McKay. Hadamard equivalence via graph isomorphism. *Discrete Mathematics*, 27(2):213–216, 1979.

[14] Brendan D. McKay. Practical graph isomorphism. *Congr. Numer.*, 30:45–87, 1981.

[15] Brendan D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

[16] Gary 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.

[17] Takunari Miyazaki. The complexity of McKay's canonical labeling algorithm. In *Groups and computation, II (New Brunswick, NJ, 1995)*, volume 28 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 239–256. Amer. Math. Soc., Providence, RI, 1997.

[18] G. E. Moorhouse. Projective planes of small order. **http://math.uwyo.edu/moorhous/pub/planes**.

[19] N. J. A. Sloane. A library of hadamard matrices. **http://www.research.att.com/~njas/hadamard/**.

[20] G. Tinhofer and M. Klin. Algebraic combinatorics in mathematical chemistry. Methods and algorithms. III. Graph invariants and stabilization methods, 1999.

[21] Jacobo Torán. On the hardness of graph isomorphism. *SIAM J. Comput.*, 33(5):1093–1108, 2004.

[22] W. D. Wallis, editor. *Computational and constructive design theory*, volume 368 of *Mathematics and its Applications*. Kluwer Academic Publishers Group, Dordrecht, 1996.