Math.comput.sci. *Online First*
© 2008 Birkhäuser Verlag Basel/Switzerland
DOI 10.1007/s11786-008-0052-8

Mathematics in
Computer Science

Automatic Proof of Graph Nonisomorphism

Arjeh M. Cohen, Jan Willem Knopper and Scott H. Murray

Abstract. We describe automated methods for constructing nonisomorphism proofs for pairs of graphs. The proofs can be human-readable or machine-readable. We have developed an experimental implementation of an interactive webpage producing a proof of (non)isomorphism when given two graphs.

Mathematics Subject Classification (2000). 05C60, 68R10, 05C25, 03B35.

Keywords. Graphs, groups, graph isomorphism, automatic proof generation.

1. Introduction

With the growth in computer power and internet access, an increasing number of problems are solved on remote machines by programs written by experts in a particular field. In this situation, the user may have no knowledge of the algorithm used, its implementation, or indeed how the remote machine is maintained. A mere yes-or-no answer cannot be trusted; we need additional verification that the answer is correct. For mathematical problems, the most obvious form of verification is a proof of correctness.

In this article, we discuss the construction of proofs for the graph isomorphism problem. Here a graph is understood to be a finite undirected graph without loops (i.e., edges having a single end point) and without multiple edges. If two graphs are isomorphic, and we are given an explicit isomorphism σ, then it is straightforward to verify that σ is an isomorphism and hence that the graphs are isomorphic. Proving that a pair of graphs are *not* isomorphic is more difficult. We show how to generate such a proof automatically.

The proof produced by our software is human-readable but could be modified to give machine-readable proofs as in [6]. We use a lot of computer time to find a short and understandable proof. Hence it can take much longer to generate a proof than to determine nonisomorphism. We discuss an experimental implementation of an interactive webpage producing such a proof when given two graphs. This works as a proof assistant in the sense that the author is able to choose options for generating such a proof. At the same time, the key ingredients of our proof are machine-readable and so can be used to construct a formal proof. In this manner, our work acts as an oracle providing the key data for a formal proof of nonisomorphism for two given graphs. This can lead to the *skeptical* use of our output by a proof checker in the terminology of [9] and [3]. Alternatively, it can be seen as a little engine of proof in the terminology of [21].

The first step in finding a non-isomorphism proof consists of a search for distinguishing invariants. An invariant is a function on graphs that takes the same value on isomorphic graphs, but may take different values on nonisomorphic graphs. In many cases, invariants give short and easy-to-verify proofs of non-isomorphism. For example, two graphs with different numbers of vertices clearly cannot be isomorphic, so the number of vertices is an easily-checked invariant. For a given input of two graphs, a distinguishing invariant is an invariant with distinct values on the two graphs. When such an invariant is found (and proven correct), the proof of nonisomorphism will be no longer than the evaluation of this function on each graph. The invariants incorporated in our software are discussed in Section 2.

If no simple invariants can be found to distinguish two graphs, we resort to general graph-isomorphism algorithms building on the methods of [6]. We have implemented Luks' algorithm [13], and modified it to output a human-readable proof. We have also modified the nauty implementation [14] of McKay's algorithm [16] to produce such a proof. In this paper we only discuss McKay's algorithm, as it gave a shorter proof than Luks' in every case we tried. Our modified version can also prove the correctness of the identification of the automorphism group of a graph. This is the content of Section 3.

We have developed an experimental proof constructor for graph nonisomorphism [20]; it is described in Section 4. It will automatically construct a proof of (non)isomorphism. In addition, a user can compose a proof interactively by choosing invariants or calling one of the modified algorithms. The proof constructor, with installation instructions, can be found in the MathDox repository [20].

Although we do not focus on complexity in this paper, it may be worth mentioning that graph nonisomorphism is neither known nor believed to be in NP, that graph isomorphism has time complexity $O(\exp(n^{1/2+o(1)}))$ (cf. [1, 13]), and that graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses (cf. [12]), where n refers to the number of vertices of G and subexponential is understood to mean $O(\exp(n^{\epsilon}))$ for every $\epsilon > 0$. The proofs that our program generates are not based on these advanced algorithms.

2. Invariants

As a first approach to finding a nonisomorphism proof for two given graphs, the following 16 invariants are checked in order.

1. number of vertices
2. number of edges
3. degree multiset

4. diameter
5. girth
6. distance multiplicity
7. local component number multiset
8. extended local component number multiset
9. characteristic polynomial of the adjacency matrix
10. Smith normal form of the adjacency matrix
11. powers of the adjacency matrix
12. multisets of numbers of triangles per vertex/edge
13. multiset of numbers of $K_{2,1,1}$-graphs per edge
14. edge distance multiplicity
15. multiset of all local $K_{2,1,1}$-graph numbers
16. multiset of all local adjacency matrix powers

Let G be a graph. The complexity of computing the invariants listed is most frequently expressed in n, the number of vertices of G, that is, invariant (1). Clearly, computation of the invariant (2), the number of edges, requires at most $\binom{n}{2}$ operations. By counting, for each vertex, the number of edges on which it lies, we find (3), the *degree multiset* $\{0, \dots, n-1\} \to \mathbb{N}$ (we use the convention $0 \in \mathbb{N}$) that assigns to each possible degree the number of vertices of G having that degree. The *distance* in G is the function on pairs of vertices that gives the shortest length of a path from one vertex to the other; here, a path of length k from v to w is a sequence $v_0, v_1, v_2, \dots, v_k$ of $k+1$ vertices of G with v_i adjacent to v_{i-1} for $i \in \{1, \dots, k\}$ and $v = v_0$, $w = v_k$; if there is no such path from v to w in G, the distance between v and w is set to ∞. Invariant (4), the *diameter* of G, is the largest distance realized in G.

Most of our algorithms assume that the graphs are *connected*, which means that the diameter is finite (see the text before the example of Section 4). The largest connected subgraphs of G are called *connected components*. The *girth* of G, invariant (5), is the smallest positive length of a path from a vertex to itself occurring in G and without repeats and retreats (so $v_{i-2} \neq v_i \neq v_{i-1}$ for each i if the path is $v = v_0, v_1, v_2, \dots, v_k = v$); if there are no such paths, the graph is a union of trees and the girth is set to ∞.

The *distance multiplicity* of a vertex is the multiset of distances between the vertex and all other vertices. The multiset of distance multiplicities of all vertices is the *distance multiplicity* of the graph, invariant (6). The *distance multiplicity* of an edge is the multiset of distances between other vertices and a vertex of that edge. The *edge distance multiplicity* of a graph, invariant (14), is the multiset of distance multiplicities of all edges. The subgraph induced on the set of vertices at distance one from a vertex is called the *neighbor graph* of that vertex. The *local component number of a vertex* is the multiset of the sizes of the connected components of this subgraph. The multiset of local component numbers of all vertices of G is the *local component number multiset*, invariant (7), of the graph. The *extended local* *component number multiset*, invariant (8), is like the local component number, except now the vertices at distance i are used instead of those at distance 1 for each distance i occurring in G.

The *adjacency matrix* A of G is an $n \times n$ matrix whose rows and columns are indexed by the vertices of G and in which all entries are 0 except those whose row and column are indexed by vertices v and w, respectively, for which (v, w) is an edge of G; for these edges the entry is 1. In particular, the trace of the adjacency matrix is equal to 0 and the (v, w)-entry $(A^k)_{vw}$ of its k-th power is nonzero if and only if there is a path of length k from v to w. Since it takes $O(n^3)$ operations to compute each power in turn, computing all distances requires at most $O(n^4)$ integer operations. The set of powers of the adjacency matrix itself is not an invariant, but it can easily be turned into one: the multiset of multisets $\{((A^1)_{vw}, \ldots, (A^{n-1})_{vw}) \mid w \text{ vertex of } G\}$ for v running over the vertices of G. This is invariant (11). Also, the *characteristic polynomial of the adjacency matrix*, invariant (9), can be computed in this time. The *Smith normal form of the adjacency matrix*, invariant (10), is stronger, i.e., distinguishes more graphs, than the rank of the adjacency modulo p for any prime p; see [10] for its precise definition and efficient computation.

A path v, u, w, v in G of length 3 with $u \neq v \neq w \neq u$ is called a *triangle* of G. If G has a triangle, its girth is 3. Invariant (12), the *multiset of numbers of triangles per vertex, edge*, counts the number of triangles on each vertex, respectively, edge. A subgraph of G on four vertices such that all unordered pairs except for one are edges is called a $K_{2,1,1}$*-graph*. Its two vertices of degree 3 form an edge that we call its *diagonal*. Counting, for each edge of G, the number of $K_{2,1,1}$-graphs having that edge as its diagonal, we obtain invariant (13), the *multiset of numbers of* $K_{2,1,1}$*-graphs per edge*. Invariant (15), the *multiset of all local* $K_{2,1,1}$*-graph numbers*, is the multiset of the values of invariant (13) for all neighbor graphs of vertices of the graph. Invariant (16), the *multiset of all local adjacency matrix powers*, is the multiset of the values of the invariant (11) for all neighbor graphs of vertices of the graph.

The order of the invariants is chosen to balance understandability with ease of calculation. In larger graphs some of the invariants further down the list become harder to verify by hand, but still can give information about the graph in polynomial time. In fact, all except invariants (10), (15), and (16) are $O(n^4)$, where n is the number of vertices of G, but some have a better exponent.

Note that some invariants are straightforward to calculate but harder to prove correct. Some effort is made to reduce the output; for example if the number of vertices with a certain degree differs in two graphs, it is not necessary to list the number of vertices with other (fixed) degrees.

3. McKay's algorithm

The current implementation of McKay's algorithm [15, 16], called nauty [14], is one of the most efficient practical graph isomorphism solvers available. We have modified this program to give additional output, that allows us to construct a human-readable proof. In this section, we outline McKay's program; in the next section, we will discuss our modifications.

Nauty's default routine for establishing nonisomorphism involves computing a *canonical labeling* for each graph. That is, a labeling of the vertices by integers with the property that two graphs are isomorphic if and only if this labeling induces an isomorphism by mapping each vertex of one graph to the vertex with the same label of the other graph. The problem with using this approach for constructing a formal proof is the definition of the canonical labeling, which is almost as involved as the algorithm itself.

We chose instead to prove nonisomorphism by constructing automorphism groups. A disadvantage of using the automorphism group is that a new graph must be constructed from the two earlier graphs and that the resulting graph is roughly the size of the union of the original graphs. Crucial to the algorithm is the notion of a colored graph (G, π), where G is a graph with vertex set V and π is an ordered partition of V (for a detailed definition, see below under *Partitions*). Note that an uncolored graph G can be interpreted as a colored graph (G, π) by taking π to be the partition $[V]$.

Before discussing the actual algorithm we introduce several useful notions.

Connection. Let (G', π') and (G'', π'') be connected colored graphs with vertex sets V' and V'', respectively, such that $V' \cap V'' = \emptyset$. Take $v' \in V'$ and $v'' \in V''$. We create a new colored graph G by adding the edge $\{v', v''\}$ to the edges of G' and G'' and creating a new partition π on $V' \cup V''$ that colors V' according to π' and V'' according to π'', except for v' and v'' which are given a new color different from all other colors. We call the colored graph (G, π) the *connection* of G' and G'' along v' and v''.

By way of example, we will take along the graphs G' and G'' pictured at the right and the left of Figure 1, respectively. They are clearly nonisomorphic and this can be proven by various of the invariants listed in Section 2. We will ignore this fact here and use them to illustrate the nauty program and our modifications. Take v' to be the vertex labeled 5 in G' and v'' the vertex 1 of G''. The connection (G, π) along 5 and 1 has vertex set $\{1, \ldots, 8\}$, edges as indicated in Figure 2, and partition π consisting of $\{1, 5\}$ and $\{2, 3, 4, 6, 7, 8\}$.

Graph isomorphism by means of connection. We can now determine whether there is an isomorphism $(G', \pi') \to (G'', \pi'')$ that takes v' to v'', by running the automorphism algorithm to compute the group $\mathrm{Aut}(G, \pi)$ of automorphisms of the connection (G, π) of (G', π') and (G'', π'') along v' and v''. Such automorphisms leave the edge $\{v', v''\}$ fixed. If $\mathrm{Aut}(G, \pi)$ contains a generator that exchanges v' and v'', then the restriction to V' of that element is an isomorphism from (G', π') to (G'', π'').

Fix a vertex $v' \in V'$. Suppose that σ is an isomorphism from (G', π') to (G'', π''). Let $v'' \in V''$ be the image of v' under σ. We will use right actions and exponentiation to denote the image, so $v'' = (v')^\sigma$. Now construct the connection

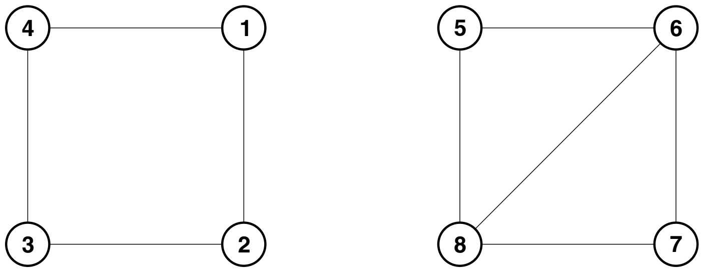


Figure 1. The graphs G'' (left) and G' (right).

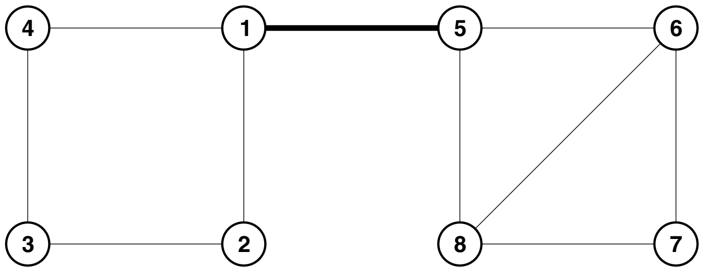


Figure 2. The connection of G' and G'' along 5 and 1.

(G, π) along v' and v''. If the automorphism algorithm is called with input (G, π), then σ can be retrieved from $\mathrm{Aut}(G, \pi)$. So, if an isomorphism $\sigma : (G', \pi') \to (G'', \pi'')$ exists, then for some $v'' \in V''$, the automorphism group of the connection (G, π) along v' and v'' must contain an element that carries v' to v''. If we can establish that, for all choices $v'' \in V''$, the automorphism group of the connection (G, π) along v' and v'' does not contain such an element, then we have a proof that the colored graphs (G', π') and (G'', π'') are not isomorphic.

If we know automorphisms of (G'', π''), then we can use these to reduce the number of checks as we now explain. Suppose that τ is an automorphism of (G'', π'') that does not fix $v'' \in V''$, so there is a vertex $u'' = (v'')^\tau$ distinct from v''. Construct the connection (G, π) along v' and v'' and calculate its group of automorphisms A. Now the group of automorphisms of the connection along v' and u'' is $B = \{\tau\sigma\tau^{-1} \mid \sigma \in A\}$. It is easily seen that the number of automorphisms that transform v' to v'' in A is equal to the number of automorphisms that transform v' to u'' in B. This means that for a nonisomorphism proof it suffices to prove the nonexistence only for one vertex v'' in each orbit under a group of known automorphisms of (G'', π''). The determination of automorphisms and orbits can easily be accompanied by automated proof generation, see [6].

In our example, Figure 1, $\mathrm{Aut}(G'')$ is transitive on $\{1, 2, 3, 4\}$, so the choice $v'' = 1$ will suffice.

Partitions. Let G be a graph with vertex set V of size n. A *partition* of the vertex set V of G is defined as a surjective map $\pi : V \to \{1, \dots, k\}$ for some $k \in \{1, \dots, n\}$.

A set consisting of all vertices with the same color, that is, the same π value, is called a *cell*. Here the natural ordering on $\mathbb{N}$ is significant for interpreting π as an ordered partition. So, the cells are ordered according to their value under π. In order to describe a partition, we will use the notation of a list, in which the cells are separated by $\,|\,$. For example, if $V = \{1, \ldots, 4\}$, then by $\pi = [1\,|\,2\,4\,|\,3]$, we mean the partition $\pi : V \to \{1, 2, 3\}$ given by $1^\pi = 1$, $2^\pi = 4^\pi = 2$, and $3^\pi = 3$. For example, the partition π of the connection of Figure 2 is denoted $[1\,5\,|\,2\,3\,4\,6\,7\,8]$.

A partition is called *discrete* if all vertices have a different color; for example $[1\,|\,4\,|\,2\,|\,3]$ is discrete. Let π and π' be partitions of a set of vertices V. Then π is called *finer* than π' if every cell of π is a subset of a cell of π' and $v^{\pi'} > v'^{\pi'} \Rightarrow v^\pi > v'^\pi$. Note that π is finer than itself. If π is finer than π' and $\pi \neq \pi'$ then π is called *strictly finer* than π'. For example, $[1\,|\,4\,|\,2\,|\,3]$ is strictly finer than $\pi = [1\,|\,2\;4\,|\,3]$.

We will use the obvious but convenient fact that, once V is identified with $\{1, \ldots, n\}$, a discrete partition of V is a permutation of $\{1, \ldots, n\}$. For example, the partition $[1\,|\,4\,|\,2\,|\,3]$ is nothing but the permutation $(2, 4, 3)$, when written in cycle notation.

Refinement function. Let G be a graph and $\pi = [V_1| \ldots |V_k]$ a partition of its vertex set V, so V_i is the inverse image under π of i for each $i \in \{1, \ldots, k\}$. For a sequence α of distinct cells of π, let $\mathcal{R}(G, \pi, \alpha)$ be a partition of V, with the following two properties.

1. $\mathcal{R}(G, \pi, \alpha)$ is finer than π.
2. $\mathcal{R}(G^\sigma, \pi^\sigma, \alpha^\sigma) = \mathcal{R}(G, \pi, \alpha)^\sigma$, for all $\sigma \in \mathrm{Sym}(V)$.

A function $\mathcal{R}$ with these properties is called a *refinement function*. Observe that, if σ is an automorphism of the colored graph (G, π), it leaves invariant all cells of π and hence α, so, by (2), $\mathcal{R}(G, \pi, \alpha)^\sigma = \mathcal{R}(G, \pi, \alpha)$, and σ is also an automorphism of $(G, \mathcal{R}(G, \pi, \alpha))$. As discrete partitions are closer to candidate automorphisms of the colored graph (in a sense to be made concrete in the context of search trees below), the intent is to refine π as much as possible.

Refinement algorithm. In Algorithm 1, we give an example of a refinement function that is part of nauty; cf. Algorithm 1 from [15] and Algorithm 2.5 in [16]. This function works well in the generic case. For certain special classes of graphs like those with high symmetry, other refinement functions might give better results, cf. [2, 14, 24]. The idea behind the algorithm is to look at the number of edges between cells of a partition. Let V_k be a cell of the partition π and let V_m be a cell occurring in α. On Line 15 of Algorithm 1, V_k appears as $\tilde{\pi}[k]$ and $\tilde{\alpha}[m]$ is $V_{m'}$ where m' is defined by the property that $\tilde{\pi}[m'] = \tilde{\alpha}[m]$. For each $v \in V_k$, we calculate the number of vertices in V_m that are adjacent to v in G. If this value is not the same for all vertices in V_k, then we can refine π to a partition $\tilde{\pi}$, in which V_k is split according to the different values. Let X be the first (that is, with the smallest $\tilde{\pi}$ value) cell of $\tilde{\pi}$ among the largest cells of $\tilde{\pi}$ contained in V_k. On Line 18 of Algorithm 1, X appears as $\pi'[t]$. The splitting is then also used to transform α to a sequence $\tilde{\alpha}$ in which the refinement is captured by adding all cells arising

Algorithm 1 Refinement algorithm

Input: G is a graph, π is the partition that needs to be refined, and α is a list of cells of π.

Returns: $\tilde{\pi}$, a partition finer than π, with $\mathrm{Aut}(G, \tilde{\pi}) = \mathrm{Aut}(G, \pi)$

```
function $\mathcal{R}(G, \pi, \alpha)$
    var
        $\tilde{\pi}$: partition                         ▷ a partition finer than $\pi$
        $\pi'$: partition                                ▷ a partition finer than $\pi$
        $\tilde{\alpha}$: sequence                       ▷ a sequence of cells of $\tilde{\pi}$
        $m$: integer                                     ▷ index of $\tilde{\alpha}$
        $t$: integer                                     ▷ position in $\pi'$
    end var
    $\tilde{\pi} := \pi$                                 ▷ $\tilde{\pi}$ is finer than $\pi$
    $\tilde{\alpha} := \alpha$                ▷ $\tilde{\alpha}$ only grows if $\tilde{\pi}$ becomes strictly finer.
    $m := 1$
    while $m \leq \mathrm{Length}(\tilde{\alpha})$ and $\tilde{\pi}$ is not discrete do
        $k := 1$
        while $k \leq \mathrm{Length}(\tilde{\pi})$ do
            $\pi' :=$ the partition of $\tilde{\pi}[k]$ induced by adjacencies with $\tilde{\alpha}[m]$
            $t :=$ the index of the first set in $\pi'$ with maximal size
            if $\tilde{\pi}[k] = \tilde{\alpha}[j]$ for some $j$ then
                replace $\tilde{\alpha}[j]$ by $\pi'[t]$
            end if
            for $i := 1$ to $t - 1$ do
                append $\pi'[i]$ to $\tilde{\alpha}$
            end for
            for $i := t + 1$ to $\mathrm{Length}(\pi')$ do
                append $\pi'[i]$ to $\tilde{\alpha}$
            end for
            refine $\tilde{\pi}$ by splitting the cell $\tilde{\pi}[k]$ into the cells of $\pi'$
            $k := k + 1$
        end while
        $m := m + 1$
    end while
    return $\tilde{\pi}$
end function
```

from V_k but X and, in case V_k occurs in α, by replacing it by X. This describes Lines 17–25 of Algorithm 1. The splitting procedure is iteratively carried out for all cells V_k of π and V_m in α after π and α have been updated with $\tilde{\pi}$ and $\tilde{\alpha}$, until no further splitting occurs.

For an example, see Figure 3. Here the partition $\pi = [1\,5\,|\,2\,3\,4\,6\,7\,8]$ of the vertex set of the graph of Figure 2 occurs at the top, and is refined to

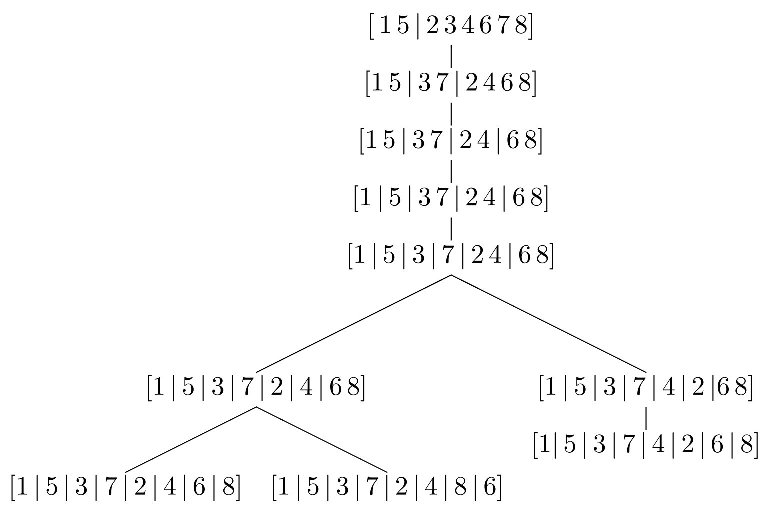


FIGURE 3. The search tree in McKay's algorithm.

[1 | 5 | 3 | 7 | 2 4 | 6 8] with parameter $\alpha = \pi$. The latter partition can be found at the fourth level below π in the tree of Figure 3. The partitions produced at intermediate steps of the refinement procedure are also given, and arise as we now explain. If we look at the adjacencies between the cells 2 3 4 6 7 8 and 1 5 we see that 3 and 7 are the only two vertices that have no connection to 1 5 and we can therefore split the partition to [1 5 | 3 7 | 2 4 6 8]. Now we look at the adjacencies between 2 4 6 8 and itself. The vertices 6 and 8 are connected to another vertex in 2 4 6 8 but 2 and 4 are not. The partition can now be split further to [1 5 | 3 7 | 2 4 | 6 8]. Now we consider the adjacencies between 1 5 and 2 4. The vertex 1 is connected to 2 4, but 5 is not. The partition can therefore be split to [1 | 5 | 3 7 | 2 4 | 6 8]. Finally we look at 3 7 and 2 4. The vertex 3 is not connected to the vertices 6, 8 and 7 is connected to both. So we end with the partition $\pi_1 =$ [1 | 5 | 3 | 7 | 2 4 | 6 8].

The search tree. Let (G, π_0) be a colored graph, with vertex set V. We define the *search tree* $T(G, \pi_0)$ whose nodes are partitions of V finer than π_0, which is its root. If π is a node of the search tree and $\rho := \mathcal{R}(G, \pi, \pi)$, with $\mathcal{R}$ as in Algorithm 1, is strictly finer than π, then ρ, or an intermediate step in the computation of ρ, will be the only child of π. Each node in the search tree representing a discrete partition will be a leaf. Let the partition $\pi = [V_1| \dots |V_k]$ be a node in the tree that is not discrete. For $v \in V$, let i be the index with $v \in V_i$ and write $\pi \circ v = [V_1| \dots |V_{i-1}|\{v\}|V_i \setminus \{v\}|V_{i+1}| \dots |V_k]$ if $|V_i| > 1$, and π otherwise. If $\pi = \mathcal{R}(G, \pi, \pi)$, then the children of π are the partitions $\pi \circ v$ for each v in the first cell of π of maximal size; we refer to this process as *case distinction*.

By way of example, part of the search tree $T(G, \pi)$ for the colored graph (G, π) of Figure 2 is given in Figure 3. At the root the coloring is the partition

$\pi_0 = \pi$ with color 1 for $\{1, 5\}$ and color 2 for $\{3, 4, 6, 7, 8\}$. The tree does not branch out before $\pi_1 = [1 \,|\, 5 \,|\, 3 \,|\, 7 \,|\, 2\,4 \,|\, 6\,8]$. As mentioned earlier, the partitions above π_1 are intermediate steps of the computation $\pi_1 = \mathcal{R}(G, \pi, \pi)$.

The search tree $T(G, \pi_0)$ can be used to compute the automorphism group of a colored graph (G, π_0) as follows. Fix a discrete partition p of V occurring as a leaf in the search tree. A second leaf p' of the search tree then determines the bijection $p^{-1}p'$ of V, which preserves the least common partition of p and p' in $T(G, \pi_0)$. By checking whether this bijection is an automorphism of (G, π_0) for each leaf p', the automorphism group can be determined elementwise.

Clearly, checking all discrete partitions is not efficient. The refinement built into the search tree is already of help in that it restricts the search to those permutations that respect given refinements of the partition induced by the initial coloring π_0. Furthermore it suffices to construct a set of generators of the automorphism group rather than all of its elements. The generating automorphisms already found can be used to reduce the number of children in the search tree that need to be perused. For, the choices of the vertex v in the case distinction above can be restricted to representatives of the orbits of the group of automorphisms found within the cell that is being split. Algorithm 2 exhibits the code involved. The list of generators found is stored in R and the subgroup of $\mathrm{Aut}(G, \pi_0)$ generated by R is denoted $\langle R \rangle$.

In general, it is not necessary to carry out the full refinement procedure. It suffices to restrict to steps in which the partitions are made finer in parallel ways as follows: if $\rho = [U_1| \cdots |U_k]$ and $\sigma = [V_1| \cdots |V_k]$ are two nodes on the same level of the search tree and there are U_i and U_j such that the number of vertices in U_j adjacent to v is not the same for each $v \in U_i$, then also the number of vertices in V_j adjacent to v is not the same for each $v \in V_i$. For, if p refines ρ and p' refines σ, then, for $p^{-1}p'$ to be an automorphism of (G, π), it must map ρ to σ.

For our example graphs of Figure 1, consider once more the search tree as given in Figure 3. Further nodes are formed by refinement and case distinction. Since the partition $\pi_1 = [1 \,|\, 5 \,|\, 3 \,|\, 7 \,|\, 2\,4 \,|\, 6\,8]$ cannot be split further by refinement (it is not necessary to prove this), the children of the corresponding node are found by case distinction of $2\,4$: we can color $2\,4$ by a partition π refining the node so that $2^\pi < 4^\pi$ or so that $4^\pi < 2^\pi$. In the left branch we have a case distinction again for the cell $6\,8$ and we find the first two leaves. The first leaf is $[1 \,|\, 5 \,|\, 3 \,|\, 7 \,|\, 2 \,|\, 4 \,|\, 6 \,|\, 8]$ which is our reference partition p, and so gives the identity map on the vertex set of G. The second leaf is $p' = [1 \,|\, 5 \,|\, 3 \,|\, 7 \,|\, 2 \,|\, 4 \,|\, 8 \,|\, 6]$, that is, the permutation $(2, 5)(4, 7, 8, 6)$, and leads to the automorphism $p^{-1}p' = (6, 8)$. Now we return to the case $4^\pi < 2^\pi$. Since we know that 6 and 8 are in the same orbit under permutations that stabilize 2 and 4, we may assume $6^\pi < 8^\pi$. This gives us another leaf $p' = [1 \,|\, 5 \,|\, 3 \,|\, 7 \,|\, 4 \,|\, 2 \,|\, 6 \,|\, 8]$, in cycle notation $(2, 5, 4, 7, 6)$, resulting in the automorphism $p^{-1}p' = (2, 5)(4, 6, 7)(2, 5, 4, 7, 6) = (2, 4)$. The leaves of the search tree are now exhausted, so the automorphism group of G is $\langle (2, 4), (6, 8) \rangle$. There are no automorphisms that interchange 1 and 5, and therefore the graphs G' and G'' are not isomorphic.

Algorithm 2 Finding generating automorphisms

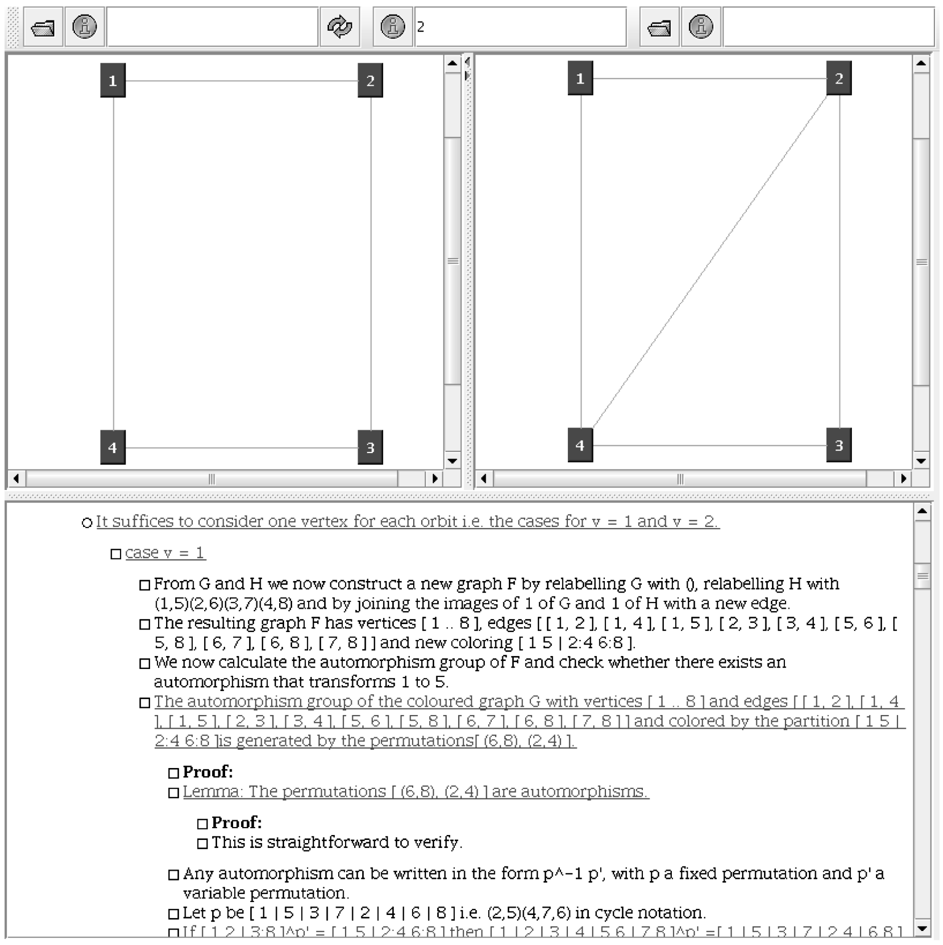
Input: (G, π) is a colored graph, p is a discrete partition finer than π, and α is a list of cells from π used to refine π.
Returns: R is a set of generators of $\mathrm{Aut}(G, \pi)$.

```
function FindAutomorphisms($G, p, \pi, \alpha$)
    var
        $c$: cell                         ▷ $c$ is the first cell of $\pi$ of maximal length
        $v$: vertex                                                          ▷ $v \in c$
        $\rho$: partition                                            ▷ $\rho$ is finer than $\pi$
        $M$: set of vertices                              ▷ used to mark vertices in $c$
    end var
    $R := \emptyset$
    $M := \emptyset$
    $\rho := \mathcal{R}(G, \pi, \alpha)$
    if $\rho$ is discrete then                          ▷ $\rho$ defines a permutation
        if $p^{-1}\rho \in \mathrm{Aut}(G, \pi)$ then                              ▷ $p' = \rho$
            $R := \{p^{-1}\rho\}$
        end if
    else                                                                ▷ recursion
        $c :=$ the first cell of $\rho$ of maximal length
        for $v \in c$ do
            if $v^\sigma \in M$ for some $\sigma$ in $\langle R \rangle$ with $\rho^\sigma = \rho$ then
                do nothing                         ▷ no new generator will be found
            else
                $R := R \cup$ FindAutomorphisms($G, p, \rho \circ v, [\{v\}]$)
                $M := M \cup \{v\}$
            end if
        end for
    end if
    return $R$
end function
```

4. A proof constructor

In this section, we report on a software package that automatically constructs a proof of (non)isomorphism of two given graphs.

The user interface. We have built an applet for a proof-of-concept webpage. It shows how a proof can be constructed interactively. The graphs can be entered in the webpage or by the user. The user can then choose to check for invariants (and select invariants from the list of fifteen in Section 2) or to request a proof that uses Luks' or McKay's algorithm and that will likely be very long; see below. Figure 4 gives a view on the applet showing part of a proof of nonisomorphism.



- It suffices to consider one vertex for each orbit i.e. the cases for v = 1 and v = 2.
  - case v = 1
    - From G and H we now construct a new graph F by relabelling G with (), relabelling H with (1,5)(2,6)(3,7)(4,8) and by joining the images of 1 of G and 1 of H with a new edge.
    - The resulting graph F has vertices [1 .. 8], edges [[1, 2], [1, 4], [1, 5], [2, 3], [3, 4], [5, 6], [5, 8], [6, 7], [6, 8], [7, 8]] and new coloring [1 5 | 2:4 6:8].
    - We now calculate the automorphism group of F and check whether there exists an automorphism that transforms 1 to 5.
    - The automorphism group of the coloured graph G with vertices [1 .. 8] and edges [[1, 2], [1, 4], [1, 5], [2, 3], [3, 4], [5, 6], [5, 8], [6, 7], [6, 8], [7, 8]] and colored by the partition [1 5 | 2:4 6:8]is generated by the permutations[(6,8), (2,4)].
      - **Proof:**
      - Lemma: The permutations [(6,8), (2,4)] are automorphisms.
        - **Proof:**
        - This is straightforward to verify.
      - Any automorphism can be written in the form p^-1 p', with p a fixed permutation and p' a variable permutation.
      - Let p be [1 | 5 | 3 | 7 | 2 | 4 | 6 | 8] i.e. (2,5)(4,7,6) in cycle notation.
      - If [1 2 | 3:8]^p' = [1 5 | 2:4 6:8] then [1 | 2 | 3 | 4 | 5 6 | 7 8]^p' =[1 | 5 | 3 | 7 | 2 4 | 6 8]

FIGURE 4. Isomorphism proof using McKay's algorithm.

The graphical frontend of our proof constructor is written in Java. Most of the algorithms are written in GAP. From Java it is possible to call these through the RIACA GAP Service by the corresponding RIACA GAP Link [8]. From GAP a modified local copy of dreadnaut is called on demand; see below for a discussion of it. The information to dreadnaut is sent in the format used by dreadnaut. The information sent back to GAP is sent in a simple XML format and parsed using the XML parser in the GAPDoc package. The resulting tree is then traversed recursively and transformed into a human-readable proof. Some of the calculations in the refinement function turn out not to be necessary in the final proof; these calculations are omitted.

The proof generated this way has a highly recursive nature. It has a hierarchical structure, with many small lemmas; see the example below. It is possible to hide the proofs of certain lemmas to take into account the different levels of mathematical expertise among users. The user can also click on a hidden part of the proof to reveal it.

For the link between the webpage and GAP we use the RIACA OpenMath library [19] and GAP phrasebook. An experimental version of the webpage can be visited at the MathDox repository [20], where the source code of the JAVA applet, the compiled JAVA applet, the GAP package (including the algorithms of [6]), and the modified nauty (for Version 2.2) can also be found.

Modifications to McKay's implementation. McKay's implementation of nauty [14] is in C [11]. Included in the implementation is an interactive program called dreadnaut. It has options to give more information. We have extended these options so that, with new options turned on, dreadnaut will produce output needed to construct a proof. In particular, for each node of the search tree for a colored graph (G, π), we discern four possibilities, where p is a fixed end node of the search tree constructed with priority.

- If it is a leaf, then the node represents a permutation p' and it is checked whether or not the product $p^{-1}p'$ is an automorphism of (G, π). The accompanying argument is easy.
- If Algorithm 1 leads to a further refinement of the partition represented by the node, the refinement step is translated into an argument why any automorphism preserving the original partition also preserves the refinement.
- If Algorithm 1 does not lead to a further refinement of the partition represented by the node, a comparison is made with a partition at the same level of which p is a refinement. If these do not match, then an argument is given why there is no automorphism preserving the given partition.
- If Algorithm 1 does not lead to a further refinement of the partition represented by the node, and the node cannot be considered a terminal node as in the previous step, a case distinction is introduced in the proof. The number of cases is diminished by referral to the orbits on vertices of automorphisms of (G, π) already found.

The resulting software can derive the automorphism group of a single graph by calling Algorithm 2. Furthermore, it can generate a proof of graph nonisomorphism of two given graphs by a run on their connections. It then creates texts with mathematical expressions which are read in from the algorithm's output by the methods of [6]. It is possible to shorten the proof by removing proofs of low-level lemmas from the output.

The proof constructor and most of the algorithms assume that the graphs are connected. However it is relatively easy to reduce proofs of nonisomorphism of unconnected graphs to nonisomorphism of its connected components.

Example. The following text shows a version of the human-readable output constructed as indicated above. For the sake of presentation, some parts have been replaced by a comment, and some minor editing has been carried out hy hand. On Lines 102 and 117, the generators of $B = \mathrm{Aut}(G, \pi)$ are found. Lines 41–81 are occupied with refinement steps. Case distinctions are initiated at Lines 82 and 88.

In comparison with the earlier treatment of this example, in Section 3, we have interchanged the roles of G' and G''. As $\mathrm{Aut}(G')$ has two orbits on $\{5, 6, 7, 8\}$, viz. $\{5, 7\}$ and $\{6, 8\}$, this entails more work than before the interchange, as $\mathrm{Aut}(G'')$ has a single orbit on the vertex set of G''. The second branch of the ensuing case division has been cut out at Line 126.

```
    Proposition (graphisomorphism): the graph K with vertices [1, 2, 3, 4] and
      edges  [[1, 2], [1, 4], [2, 3], [3, 4]] and the graph H with vertices
      [1, 2, 3, 4] and edges [[1, 2], [1, 4], [2, 3], [2, 4], [3, 4]] are not
      isomorphic.
 5  Proof:
      Suppose that p is an isomorphism that transforms K to H. Let v = 1^p. For
      all vertices v of H we show that there are no isomorphisms transforming 1
      to v.
      To prove this we use information about the orbits of H under automorphisms
10    on H. If a is an automorphism and v^a=v', then 1^p = v' if and only if
      1^p^(a^-1) = v. In other words it is enough to verify for all v in
      different orbits.
      Let A be the group generated by (2,4) and (1,3). It is straightforward to
      verify that A is a group of automorphisms of H. Then we calculate the
15    orbits.
      Proposition: The orbits of A are [1, 3] and [2, 4]. (* proof hidden *)
      case distinction: It suffices to consider one vertex for each orbit
        i.e. the cases for v = 1 and v = 2.
        case v = 1:
20        From K and H we now construct a new graph G by relabelling K with (),
          relabelling H with (1,5)(2,6)(3,7)(4,8) and by joining the images of 1
          of K and 1 of H with a new edge.
          The resulting graph G has vertices [1 .. 8], edges [[1, 2], [1,
          4], [1, 5], [2, 3], [3, 4], [5, 6], [5, 8], [6, 7], [6,
25        8], [7, 8]] and new coloring [1 5 | 2:4 6:8].
          We now calculate the automorphism group of G and check whether there
          exists an automorphism that transforms 1 to 5.
          Proposition (automorphismgroup): The automorphism group of the colored
            graph G with vertices [1 .. 8] and edges [[1, 2], [1, 4], [1, 5],
30          [2, 3], [3, 4], [5, 6], [5, 8], [6, 7], [6, 8], [7, 8]] and colored
            by the partition [1 5 | 2:4 6:8] is generated by the permutations
            [(6,8),(2,4)].
          Proof:
            Denote the automorphism group by B.
35          Lemma: The permutations [(6,8), (2,4)] are automorphisms. (This is
              straightforward to verify.)
```

```
Any automorphism can be written in the form p^-1 p', with p a fixed
permutation and p' a variable permutation.
Let p be [1 | 5 | 3 | 7 | 2 | 4 | 6 | 8] i.e. (2,5)(4,7,6) in
cycle notation.
refinement: If [1 2 | 3:8]^p' = [1 5 | 2:4 6:8] then
  [1 | 2 | 3 | 4 | 5 6 | 7 8]^p' =[1 | 5 | 3 | 7 | 2 4 | 6 8].
Proof:
  Lemma (refine part)
    If [1 2 | 3:8]^p' = [1 5 | 2:4 6:8] then [1 2 | 3 4 | 5:8]^p' =
    [1 5 | 3 7 | 2 4 6 8].
  Proof:
    Look at [1 2]^pi and how it is connected to [3:8]^pi, for
    pi=p,p'.
    First for p
      [1 2]^p = [1 5].
      [3:8]^p = [2:4 6:8].
      The vertices 3 7 are not connected to any vertices of [1 5].
      The vertices 2 4 6 8 are each connected to 1 vertex of [1 5].
    QED(p)
    Then for p'
      [1 2]^p' = [1 5].
      [3:8]^p' = [2:4 6:8].
      The vertices 3 7 are not connected to any vertices of [1 5].
      The vertices 2 4 6 8 are each connected to 1 vertex of [1 5].
    QED(p')
    If p^-1p' is an automorphism then it transfers [1 5] to [1 5]
    and [2:4 6:8] to [2:4 6:8] and must therefore transfer [3
    7] to [3 7] and [2 4 6 8] to [2 4 6 8].
    Since [1 2 | 3 4 | 5:8]^p = [1 5 | 3 7 | 2 4 6 8], we now
    know that [1 2 | 3 4 | 5:8]^p' = [1 5 | 3 7 | 2 4 6 8].
  QED(refine part)
  Lemma (refine part)
    If [1 2 | 3 4 | 5:8]^p' = [1 5 | 3 7 | 2 4 6 8] then [1 2 |
    3 4 | 5 6 | 7 8]^p' = [1 5 | 3 7 | 2 4 | 6 8].
  (* proof hidden *)
  Lemma (refine part)
    If [1 2 | 3 4 | 5 6 | 7 8]^p' = [1 5 | 3 7 | 2 4 | 6 8] then
    [1 | 2 | 3 4 | 5 6 | 7 8]^p' = [1 | 5 | 3 7 | 2 4 | 6 8].
  (* proof hidden *)
  Lemma (refine part)
    If [1 | 2 | 3 4 | 5 6 | 7 8]^p' = [1 | 5 | 3 7 | 2 4 | 6 8]
    then [1 | 2 | 3 | 4 | 5 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 2 4 |
    6 8].
  (* proof hidden *)
QED(refinement)
case distinction 5^p': Now we look at the consequences of
  [1 | 2 | 3 | 4 | 5 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 2 4 | 6 8] for
  different possibilities for 5^p'.
  Suppose that 5^p' = 2.
    Now [1 | 2 | 3 | 4 | 5 | 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 2 | 4
    | 6 8].
    case distinction 7^p': Now we look at the consequences of
```

```
       [1 | 2 | 3 | 4 | 5 | 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 2 | 4 |
       6 8] for different possibilities for 7^p'.
       Suppose that 7^p' = 6.
        Now [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]^p' = [1 | 5 | 3 | 7 |
        2 | 4 | 6 | 8].
        So p' = [1 | 5 | 3 | 7 | 2 | 4 | 6 | 8] or (2,5)(4,7,6).
        Then p^-1p' = () is an automorphism.
        Furthermore it is included in B (it is the identity).
       QED(case 7^p' = 6)
       Suppose that 7^p' = 8.
        Now [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]^p' = [1 | 5 | 3 | 7 |
        2 | 4 | 8 | 6].
        So p' = [1 | 5 | 3 | 7 | 2 | 4 | 8 | 6] or (2,5)(4,7,8,6).
        Then p^-1p' = (6,8) is an automorphism.
        Furthermore it is included in B (it is a generator of B).
       QED(case 7^p' = 8)
      QED(case distinction 7^p')
     QED(case 5^p' = 2)
     Suppose that 5^p' = 4.
      Now [1 | 2 | 3 | 4 | 5 | 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 4 | 2
      | 6 8].
      Now we look at all the consequences of [1 | 2 | 3 |
      4 | 5 | 6 | 7 8]^p' = [1 | 5 | 3 | 7 | 4 | 2 | 6 8] for
      different possibilities for 7^p'.
       Suppose that 7^p' = 6.
        Now [1 | 2 | 3 | 4 | 5 | 6 | 7 | 8]^p' = [1 | 5 | 3 | 7 |
        4 | 2 | 6 | 8].
        So p' = [1 | 5 | 3 | 7 | 4 | 2 | 6 | 8] or (2,5,4,7,6).
        Then p^-1p' = (2,4) is an automorphism.
        Furthermore it is included in B (it is a generator of B).
       QED(case 7^p' = 6)
      (* the case 7^p' = 8 is ruled out by an orbit argument. *)
      QED(case distinction 7^p')
     QED(case 5^p' = 4)
    QED(case distinction 5^p')
   QED(automorphismgroup)
  QED(case v = 1)
  case v = 2: (* is similar to the case v = 1 and hence omitted. *)
 QED(case distinction)
QED(graphisomorphism)
```

Conclusion. We have modified McKay's program nauty (or rather the dreadnaut part) so as to produce proof certificates and human-readable proofs for graph (non)isomorphism. We extended this to an experimental webpage providing a service for (non)isomorphism proofs for any pair of submitted graphs. The communication of the webpage with the modified nauty takes place via GAP. The webpage is enriched with other options for constructing nonisomorphism proofs, such as a search for distinguishing invariants.

The idea of constructing certificates is not new. Besides the work [6] already quoted on permutation groups that was used in the current research, we mention primality proofs [5], the certified Buchberger algorithm [22], and the trace option for noncommutative Gröbner bases in [7]. A major reason for focusing on graph nonisomorphism is its significance for a wide range of applications, varying from image analysis [4] and organic molecule structures [23] to designs of experiments [18].

Because of the exponential growth in the lengths of the proofs produced, our modified version of McKay's algorithm is only practical for relatively small graphs. Invariants can frequently distinguish much larger graphs, however.

We benchmarked our invariants with the series of all 3854 nonisomorphic strongly-regular graphs with parameters $(35, 16, 6, 8)$ available on McKay's webpage [17]. By definition, in such a graph there are exactly 35 vertices, each vertex has degree 16, on each edge there are precisely 6 triangles, and each pair of nonadjacent vertices has exactly 8 common neighbors. Because of their regularity, the most straightforward invariants will not separate them. Of the first fourteen invariants, (13), the multiset of numbers of $K_{2,1,1}$-graphs per edge, turned out to be the most effective distinguisher. It takes 3487 values on the graphs, whereas for instance invariant (10) only takes 5 different values. Invariant (15), the *multiset of all local $K_{2,1,1}$-graph numbers*, has 3804 different values. Using (16), the *multiset of all local adjacency matrix powers*, in addition to (15), we can distinguish them all.

The package is experimental and bigger graphs still lead to huge output. Finetuning by skipping subroutines from [6] will lead to a considerable improvement of the performance. Also the protocol and the implementation of the communication between processes leaves room for speed up. At the moment a lot of information is sent from dreadnaut to GAP. A more efficient coupling will also diminish running times.

A more mathematical approach to shortening the proofs would be to use an initial coloring of the graph. For instance, the user could introduce a vertex invariant as the coloring in the first step of McKay's algorithm. In the example of Section 4, the case $v = 2$ could be ruled out by the observation that the valence of vertex 6 in G does not match the valence of vertex 2 in G.

Adding these options to the webpage would lead to a better proof assistant in that invariants could be introduced to shorten automatically generated proofs. Also, the nauty user's guide [14] describes further refinements already implemented in nauty but not employed by us for proof generation. Furthermore, in order to be able to deal with more regular graphs, refinements based on edges, and more complicated structures than vertices, could be incorporated, as proposed in [24].

References

[1] L. Babai, Moderately exponential bound for graph isomorphism. In *Fundamentals of computation theory* (*Szeged,* 1981), vol. 117 of *Lecture Notes in Comput. Sci.*, pages 34–50. Springer, Berlin, 1981.

[2] L. Babel, I. V. Chuvaeva, M. Klin, and D. V. Pasechnik, Program implementation of the Weisfeiler-Leman algorithm. In *Algebraic Combinatorics in Mathematical Chemistry. Methods and Algorithms*, vol. TUM-M9701, pages 1–45, 1997.

[3] H. Barendregt and A. M. Cohen, Electronic communication of mathematics and the interaction of computer algebra systems and proof assistants. *J. Symb. Comput.*, 32(1–2):3–22, 2001.

[4] H. Bunke and B. T. Messmer, Efficient attributed graph matching and its application to image analysis. In *ICIAP '95: Proceedings of the 8th International Conference on Image Analysis and Processing*, pages 45–55, London, UK, 1995. Springer Verlag.

[5] O. Caprotti and M. Oostdijk, Formal and efficient primality proofs by use of computer algebra oracles. *J. Symbolic Comput.*, 32(1–2):55–70, 2001. Computer algebra and mechanized reasoning (St. Andrews, 2000).

[6] A. Cohen, S. Murray, M. Pollet, and V. Sorge, Certifying solutions to permutation group problems. In Franz Baader, editor, *19th International Conference on Automated Deduction*, pages 258–273. Springer Verlag, Berlin, 2003.

[7] A. M. Cohen, Documentation on the gbnp package, 2007. Available from World Wide Web: `http://www.mathdox.org/products/gbnp/`.

[8] RIACA GAP phrasebook, 2004. Available from World Wide Web: `http://www.mathdox.org/phrasebook/gap/`.

[9] J. Harrison and L. Théry, A skeptic's approach to combining HOL and Maple. *J. Autom. Reasoning*, 21(3):279–294, 1998.

[10] C. S. Iliopoulos, Worst-case complexity bounds on algorithms for computing the canonical structure of finite abelian groups and the Hermite and Smith normal forms of an integer matrix. *SIAM J. Comput.*, 18(4):658–669, 1989.

[11] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[12] A. R. Klivans and D. van Melkebeek, Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM J. Comput.*, 31(5):1501–1526 (electronic), 2002.

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

[14] B. D. McKay, *nauty User's Guide*. Computer Science Department Australian National University, ACT 0200, Australia. Available from World Wide Web: `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`. version 2.2.

[15] B. D. McKay, Computing automorphisms and canonical labellings of graphs. In *Combinatorial mathematics* (*Proc. Internat. Conf. Combinatorial Theory, Australian Nat. Univ., Canberra,* 1977), vol. 686 of *Lecture Notes in Math.*, pages 223–232. Springer, Berlin, 1978.

[16] B. D. McKay, Practical graph isomorphism. In *Proceedings of the Tenth Manitoba Conference on Numerical Mathematics and Computing*, vol. I, (*Winnipeg, Man.,* 1980), vol. 30, pages 45–87, 1981.

[17] Lists of nonisomorphic strongly-regular graphs, 2008. Available from World Wide Web: `http://cs.anu.edu.au/~bdm/data/graphs.html`.

[18] N. Van Minh Man, *On Construction and Identification of Graphs.* Technische Universiteit Eindhoven, Eindhoven, 2005. PhD Thesis.

[19] RIACA OpenMath library, 2004. Available from World Wide Web: `http://www.mathdox.org/omlib/`.

[20] Proof assistant for graph non-isomorphism, 2006. Available from World Wide Web: `http://www.mathdox.org/graphiso/`.

[21] N. Shankar, Little engines of proof. In *FME*, pages 1–20, 2002.

[22] L. Théry, A certified version of Buchberger's algorithm. In *CADE*-15*: Proceedings of the* 15*th International Conference on Automated Deduction*, pages 349–364, London, UK, 1998. Springer Verlag.

[23] M. I. Trofimov and N. D. Zelinsky, Application of the electronegativity indices of organic molecules to tasks of chemical informatics. *Russian Chemical Bulletin*, 54:2235–2246, 2005.

[24] B. Weisfeiler, *On Construction and Identification of Graphs*. Springer Verlag, Berlin, 1976. With contributions by A. Lehman, G. M. Adelson-Velsky, V. Arlazarov, I. Faragev, A. Uskov, I. Zuev, M. Rosenfeld and B. Weisfeiler, B. Weisfeiler (ed.), Lecture Notes in Mathematics, Vol. 558.

Arjeh M. Cohen and Jan Willem Knopper
Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
PO Box 513
NL-5600 MB Eindhoven
The Netherlands
e-mail: `amc@win.tue.nl`
`jknopper@win.tue.nl`

Scott H. Murray
Department of Mathematics and Statistics
University of Sydney
NSW 2006
Australia
e-mail: `murray@maths.usyd.edu.au`

Received: December 14, 2007.

Revised: March 14, 2008.

Accepted: June 4, 2008.