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 a proof generator for graph nonisomorphism, which allows users to input graphs and construct a proof of (non)isomorphism.

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 construct such proofs for the problem of graph isomorphism.

If two graphs are isomorphic, and we are given an isomorphism, then it is easy to prove this by checking the isomorphism. Proving that a pair of graphs are *not* isomorphic is more difficult. We show how to generate such a proof automatically. Our proofs are intended to be human-readable but could be modified to give machine-readable proofs as in [3]. 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. Although we are primarily interested in practical computations, we occasionally use the concept of polynomial-time algorithms [5, Chapter 36].

In Section 2, we look at invariants: functions that take 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 nonisomorphism. For example, two graphs with different numbers of vertices clearly cannot be isomorphic, so this is an easily-checked invariant.

When no simple invariants can be found to distinguish two graphs, we resort to general graph-isomorphism algorithms building on the methods of [4]. We have implemented the algorithm of Luks [14], and modified it to output a human-readable proof. We have also modified the nauty implementation [15] of McKay's algorithm [17] to produce such a proof. We only discuss McKay's algorith, since it gave a shorter proof than Luks' in every case we tried. The modified version of McKay's algorithm can also prove the correctness of the automorphism group of a graph.

We have developed a proof generator for graph nonisomorphism [19], described in Section 4. This will automatically construct a proof of (non)isomorphism, and can also be used to compose a proof interactively by choosing invariants or calling one of the modified algorithms. The algorithms are implemented in GAP [6], apart from the modifications to nauty, which is in C [13]. The user interface is written in Java [21]. The proof generator, with installation instructions, can be found online at [19] or in the RIACA software repository.

Date: June 22, 2007.

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.

The proofs have a hierarchical structure, with many small lemmas (see the example in ???????). It is possible to hide the proof 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.

[Mentions analysis here, including nauty not known to be poly time]

listinvariants

2. Invariants

In order to check whether two graphs are isomorphic, the following invariants are checked in order:

(1) number of vertices
(2) number of edges
(3) degree multiset
(4) diameter
(5) girth
(6) distance multiplicity
(7) subgraph invariance
(8) extended subgraph invariance
(9) characteristic polynomial of the adjacency matrix and Seidel matrix
(10) Smith normal form of the adjacency matrix
(11) powers of the adjacency matrix
(12) number of triangles per vertex, edge (multiset)
(13) number of $K_{2,1,1}$-graphs per edge (multiset)
(14) edge distance multiplicity
(15) multiset of all edge invariants per edge

The precise definitions of these invariants can be found in [2].

The order of the invariants is chosen to balance understandability with ease of calculation. In larger graphs some of the invariants high in the tree become harder to humanly verify, but still can give information about the graph.

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 needed to mention the number of vertices with a different degree.

mckay

3. McKay's algorithm

3.1. **Introduction.** The current implementation of McKay's algorithm [16, 17], called nauty [15], is one of the most efficient practical graph isomorphism solvers available. We have modified this program to give additional output, which allows us to construct a human-readable proof.

Nauty's default routine for establishing nonisomorphism involves computing a *canonical labelling* for each graph. That is, a labelling of the vertices by integers with the property that two graphs are isomorphic iff this labelling induces an isomorphism. The problem with this for constructing a formal proof is that the definition of the canonical labelling 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 twice as big as the original graphs. Let $G = (V, E, \gamma)$ and $G' = (V', E', \gamma')$ be two connected graphs. Let $v \in V$ and $v' \in V'$. We create a new graph G'' by relabeling V' so that V and V' are disjoint, adding an edge $\{v, v'\}$ and creating a new coloring function γ'' that colors the vertices in V like

γ and the vertices in V' like γ' except for v and v' which are given a new color c that is different from all other colors. In other words, $G'' = (V'', E'', \gamma'')$ where $V'' = V \cup V'$, $E'' = E \cup E' \cup \{\{v, v'\}\}$, $\gamma''(u) = \gamma(u)$ for $u \in V \setminus \{v\}$, $\gamma''(u') = \gamma'(u')$ for $u' \in V' \setminus \{v'\}$, and $\gamma''(v) = \gamma''(v') = c$.

We can now determine whether there is an isomorphism $G \to G'$ that takes v to v', by running the automorphism algorithm to compute the group of automorphisms of G'' (they leave the edge $\{v, v'\}$ fixed). If the resulting group of automorphisms contains an element that exchanges v and v' then that element gives an isomorphism between G and G'.

Now fix a vertex $v \in V$. Suppose there exists an isomorphism between G and G'. Let σ be such an isomorphism. Let $v' \in V'$ be the image of v under σ. Now construct G''. If the automorphism algorithm is called with G'', then σ can be retrieved from the automorphism group.

Suppose we want to prove that there exist no isomorphisms that transform G to G'. We can then choose $v \in V$. If an isomorphism σ exists, then for some $v' \in V$ the computed group of automorphisms of G'' must contain an element that transfers v to v'. If we can prove that for all $v' \in V'$ the automorphism group of the corresponding G'' contains no such element then this is a proof that the graphs are not isomorphic.

If we know automorphisms of G' then we can use these to reduce the number of checks. Suppose τ is an automorphism of G', but not the identity and $v' \in V'$ is not fixed under τ, then there is a $u' = \tau(v') \neq v'$. Suppose G is a graph as above and $v \in V$ fixed. Now construct G'' with v' and calculate the group of automorphisms A. Now the group of automorphisms for G'' constructed with u' is $B = \{\tau\sigma\tau^{-1} \mid \sigma \in A\}$. It is easy to see 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 is sufficient to prove the nonexistence only for one vertex in each orbit under a group of known automorphisms of G'.

3.2. **Algorithms and variables.** Let $G = (V, E)$ be a finite graph. A *partition* is defined as a vertex coloring $\pi : V \to C$ with an ordering on $\pi(V) = C$. For example, by $\pi = [1 \mid 2\ 4 \mid 3]$, we mean $\pi(1) < \pi(2) = \pi(4) < \pi(3)$. A set consisting of vertices with the same color is called a *cell*. A partition is called *discrete* if all vertices have a different color, for example $[1 \mid 2 \mid 3 \mid 4]$ 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 in π' and $\pi'(v) > \pi'(v') \Rightarrow \pi(v) > \pi(v')$; π' is then called *coarser* than π. Note that π is both finer and coarser than itself. If π is finer (or coarser) than π' and $\pi \neq \pi'$ then π is called *strictly finer* (or *strictly coarser*) than π'. The number of cells of π is denoted by $|\pi|$. Let $v \in V$ and $W \subseteq V$. Define $\mathrm{adj}_W(v)$ to be the number of elements of W which are adjacent to v in G. A partition is *equitable* (with respect to G) if for all pairs of cells $c, d \in \pi$ and $u, v \in c$, $\mathrm{adj}_d(u) = \mathrm{adj}_d(v)$.

The basic search tree. Let $G = (V, E, \gamma)$ be a colored graph. A discrete partition gives a labeling of G. With two discrete partitions of the same vertices it is possible to construct the vertex map that takes a vertex to the vertex in the second partition with the same index. It is then possible to check whether this map is an automorphism. Now let p be a discrete partition finer than $\pi_0 = \gamma(V)$. If we check for each discrete partition p' finer than π_0, whether the map between p and p' is an automorphism then we have found all automorphisms in the automorphism group of G.

Checking all discrete partitions is not efficient. Fortunately it is possible to reduce the number of checks by refinement and further it is sufficient to not generate the full automorphism group but only generate its generators. This means that known automorphisms can be used to reduce the number of possibilities. In this subsection we describe the basic search tree and the methods to reduce the number of checks.

We now define the *search tree* $T(G, \pi)$ on the nodes labeled by partitions of V. The root is π. A node in the tree with a discrete partition is a leaf. Let the partition π be a

FindAutomorphisms

Algorithm 1 Finding all isomorphisms (1)

Input: G is a graph (used to check automorphism), p is the reference discrete partition, π is a partition finer than π_0 and $\tilde{\pi}$ is the set of cells with which to refine
Returns: *result* is (the set of generators of) the group of automorphisms of G that fix π

```
1:  function FindAutomorphisms(G, p, π)
2:      var
3:          c:cell                          ▷ c is the first cell of π of maximal length
4:          v:vertex                                                    ▷ v ∈ c
5:          π′:partition                                  ▷ a partition finer than π
6:      end var
7:      π′ := R(G, π, π)
8:      result := ∅
9:      if π′ is discrete then                                  ▷ p and π′ define a map
10:         if the map from (p, π′) denotes an automorphism then
11:             result := {that automorphism}
12:         else
13:             result := ∅
14:         end if
15:     else        ▷ recursion; this terminates since the number of cells in π will increase
16:         c :=the first cell of π′ of maximal length
17:         for v ∈ c do
18:             result := result∪FindAutomorphisms(G, p, π′ ∘ v).
19:         end for
20:     end if
21:     return result
22: end function
```

node in the tree that is not discrete. Then the children of π are the partitions $\pi \perp w$ for each w in the first cell of π with maximal length.

A leaf gives a labeling of the graph. From two discrete partitions on the same set of points it is possible to construct a vertex map taking a vertex to the vertex in the second partition with the same as index as the vertex in the first partition.

By comparing all leaves with the first leaf, the complete automorphism group can be obtained.

Using an indicator function (or partition invariant). Let $G = (V, E)$ be a graph. Let ρ be the root node in a basic search tree of G with partition π_ρ. Let ν be a node in that basic search tree with partition π_ν. Let $\sigma \in \mathrm{Sym}(V)$. Let Λ be a function on all combinations of G, ρ, π_ρ and π_ν to an ordered set Δ. If Λ has the property that $\Lambda_{G^\sigma,\pi_\rho^\sigma}(\pi_\nu^\sigma) = \Lambda_{G,\pi_\rho}(\pi_\nu)$ then we call it an *indicator function* or *partition invariant*.Let $\rho = \nu_1, \ldots, \nu_k = \nu$ be the path from the root node ρ to ν and let Λ be an indicator function. We can now define another indicator function $\tilde{\Lambda}_{G,\pi}(\nu) = (\Lambda_{G,\pi}(\nu_1), \Lambda_{G,\pi}(\nu_2), \ldots, \Lambda_{G,\pi}(\nu_k))$ to the set Δ^+, with the lexicographic ordering induced by the ordering of Δ. Δ^+ is here the set $\Delta \cup \Delta \times \Delta \cup \Delta^3 \cup \cdots$.

Leaves with partitions with different values of Λ represent nonisomorphic labelings. Therefore graphs colored according to nodes with different values of $\tilde{\Lambda}$ cannot be isomorphic. At the moment none of the partition invariants used by McKay are used. We use something comparable though: we check if p and p' are refined in the same way; if they are not, they cannot result in an automorphism.

Refinement function. Let $G = (V, E)$ be a graph. Let $\pi = (V_1, \ldots, V_k)$ be a partition of V. Let $\alpha = (V_{i_1}, \ldots, V_{i_l})$ be a sequence of distinct cells of π. Let $\mathcal{R}_{G,\pi}(\alpha)$ be a partition of V, with the following properties:

FindAutomorphismsFull

Algorithm 2 Finding all isomorphisms (2)

Input: G is a graph (used to check automorphism), p is the reference discrete partition, π is a partition finer than π_0, $\tilde{\pi}$ is the set used to refine

Returns: *result* is (the set of generators of) the group of automorphisms that of G that fix π.

```
1: function FindAutomorphisms(G, p, π, π̃)
2:     var
3:         c:cell                                   ▷ c is the first cell of π of maximal length
4:         v:vertex                                                              ▷ v ∈ c
5:         π′:partition                                                ▷ π′ is finer than π
6:     end var
7:     result := ∅
8:     π′ := R(G, π, π̃)
9:     if π′ is discrete then                                          ▷ p and π′ define a map
10:        if the map from (p, π′) denotes an automorphism then
11:            result := {that automorphism}
12:        else
13:            result := ∅
14:        end if
15:    else                          ▷ recursion; this terminates since the number of cells in
16:        c :=the first cell of π′ of maximal length
17:        for v ∈ c do
18:            if an automorphism σ of G is known such that σ(π′) = π′ and such that
    there exists a marked v′ ∈ c with σ(v′) = v then
19:                do nothing      ▷ no new generator will be found, v does not have to be
    marked
20:            else
21:                result := result∪FindAutomorphisms(G, p, π′ ∘ v, (v))
22:                mark v
23:            end if
24:        end for
25:    end if
26:    return result
27: end function
```

(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 defined this way is called a *refinement function*. Now we will give an example of a refinement function (this is algorithm 1 from [16] McKay77 and algorithm 2.5 in [17] McKay81). This is the standard algorithm that is used in nauty. For some types of graphs other refinement functions might give better results.

The idea behind the algorithm is looking at the number of edges between cells of a partition. Let V_i be cells of a partition π. Let V_j be another cell of π. Now calculate the value of $\mathrm{adj}_{V_i}(\cdot)$ for the points in V_j. If the value is not the same for all points, then it is possible to make a finer partition, in which V_j is split according to the different values.

This function can be used to narrow down the number of possibilities. The number of cells can be increased in a way that is invariant under automorphisms. When using a reference discrete partition it is also possible to check if the $\mathrm{adj}_v(\cdot)$ values are the same. If they are not, then no map from a partition finer than the current partition and the reference partition can be an automorphism. This has been implemented in the proof assistant.

Define $\pi \perp v$ to be the refinement $\mathcal{R}(G, \pi \circ v, (\{v\}))$. If $|V_i| = 1$ then $\pi \circ v$ is π.

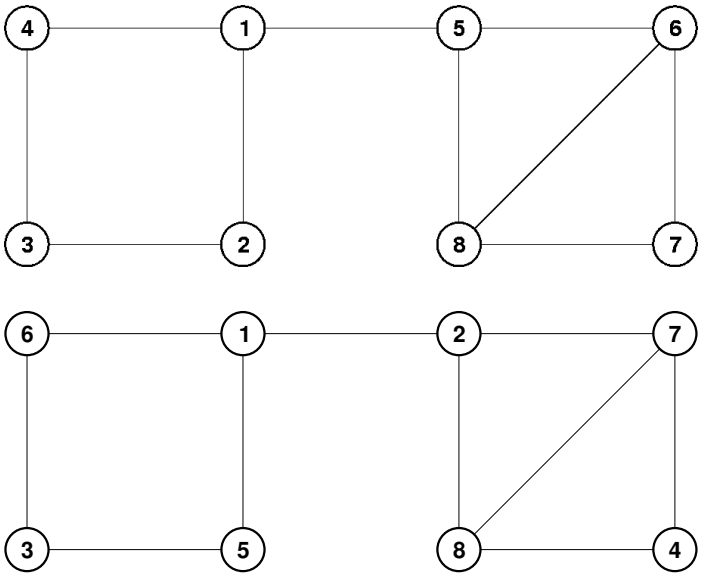

FIGURE 1. A graph and the corresponding new labeling resulting from the partition [1|5|3|7|2|4|6|8]

3.3. **Implementation and our modifications.** A pair of discrete partitions of the same graph gives a map from V to V. If such a map keeps the edges invariant it is an automorphism. Note that it is possible to generate the automorphism group by fixing one discrete partition and letting the other run through the possibilities.

These possibilities can be narrowed down by using the refinement function $\mathcal{R}$. Let π and π' be partitions of the same graph. Suppose there exists an automorphism σ such that for every vertex v $\pi(v) = \pi'(\sigma(v))$, then because of the nature of the refinement function $(\mathcal{R}(\pi))(v) = (\mathcal{R}(\pi'))(\sigma(v))$. In general it is not necessary to prove the full refinement procedure. It is enough to show that the step, in which the partitions are made finer goes parallel (if it doesn't then there cannot be an automorphism and we're finished).

For the children of the node, it is enough to look at vertices in different orbits. Suppose π is a partition in the tree and u_1 and u_2 are vertices to split and a is an automorphism such that $a(u_1) = u_2$, then for every vertex v : $a((\pi \perp u_1)(v)) = (\pi \perp u_2)(v)$. This means that the node $\pi \perp u_2$ has only leaves as descendants that are either not isomorphic or isomorphic with an automorphism already calculated from the descendants of $\pi \perp u_1$.

Each leaf, or discrete partition in the search tree, is compared with the fixed partition. If the resulting map is an automorphism it is added to the generators of the automorphism group.

McKay has written an implementation of his algorithm called nauty [15]. This implementation is in C [13]. 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, we display which node of the search tree we are currently working on. We also display the partition computed in line 15 of Algorithm 3, whenever this partition is strictly finer than the existing one.

This modified dreadnaut program is called from GAP. The data from the calculation in dreadnaut is sent to standard output in XML form and parsed using the XML parser

Refinement

Algorithm 3 Refinement algorithm

Input: G is a graph (used to calculate d), π is the partition that needs to be refined and $\alpha = (W_1 \dots W_M)$ is a list of cells, with which the partition will be refined.
Returns: $\tilde{\pi}$, a partition finer than π ▷ more can be said, but this is not needed to generate a proof

```
1:  function R(G, π, α)
2:      var
3:          π̃:partition                                   ▷ a partition finer than π
4:          π′:partition                                  ▷ a partition finer than π
5:          α̃:partition                                   ▷ a partition finer than α
6:          m:integer                                     ▷ index of α̃
7:          t:integer                                     ▷ position in π′
8:      end var
9:      π̃ := π                                            ▷ π̃ is finer than π
10:     α̃ := α
11:     m := 1                          ▷ M = α̃ only grows if π̃ becomes strictly finer.
12:     while m ≤ |α̃| and π̃ is not discrete do
13:         k := 1                ▷ Let |π̃| = K. Then K − k decreases and is nonnegative.
14:         while k ≤ |π̃| do
15:             calculate the partition π′ = (X_1, ..., X_s) of π̃[k] ordered by adj_{α̃[m]}.
16:             let t be the index of the first set in π′ with maximal size

17:             if π̃[k] = α̃[j], for any j then
18:                 replace α̃[j] by π′[t]
19:             end if

20:             for i := 1 to t − 1 do
21:                 append π′[i] to α̃
22:             end for
23:             for i := t + 1 to |π′[i]| do
24:                 append π′[i] to α̃
25:             end for

26:             update π̃ by splitting the cell π̃[k] into the cells X_1, ..., X_s in that order.
                                                          ▷ π̃ becomes finer.
27:             k := k + 1
28:         end while
29:         m := m + 1
30:     end while
31:     return π̃
32: end function
```

in the GAPDoc package. The resulting tree is then traversed recursively and transformed into a human readable proof. At the moment a lot of information is sent from dreadnaut to GAP in this way. It should be possible to reduce this to improve performance. Some of the calculations in the refinement function turn out not to be necessary in the final proof, but know beforehand that they are necessary. These calculations are removed

3.4. **Example.** We want to check whether the two graphs are isomorphic, but the part of nauty that we use gives the automorphism group of a colored graph. It is then possible to check whether a vertex can be mapped to a vertex of the other graph by creating a new graph by adding an edge between two vertices of the same color of different graphs,

coloring these two vertices in a new color and running the algorithm on that graph and that edge. It is clear that if for all pairs of vertices there are no automorphisms that exchange the graphs, there is no graph isomorphism. It is sufficient to fix a vertex in one of the graphs and to only use one vertex in an orbit of the other graph.

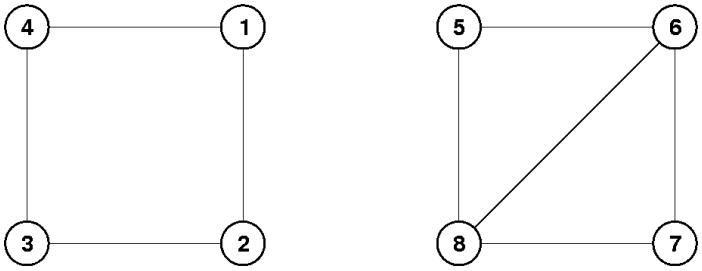

FIGURE 2. Two graphs

nauty1

Now look at the graphs in Figure 2. We use the upper left vertex of the right graph and look at the orbits of the left graph. All vertices are in the same orbit (under rotation). So it is sufficient to do the construction on the upper right vertex: see Figure 3.

nauty1

nauty2

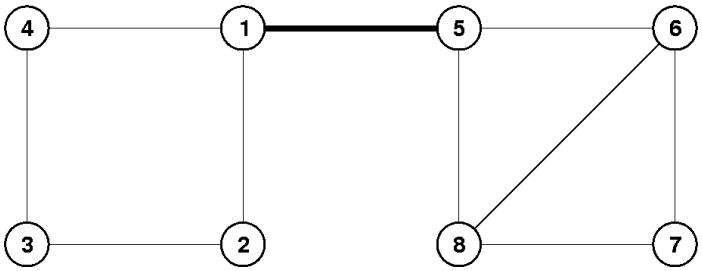

FIGURE 3. The two graphs connected by an edge

nauty2

The search tree in Figure 4 is formed by refining and case distinction. The root of the search tree is the starting partition [15|234678]. From looking ahead at the algorithm output, we get the reference partition $p = [1|5|3|7|2|4|6|8]$. The starting partition can be refined to [1|5|3|7|24|68] in a number of steps. Since there has not been case distinction yet, all discrete partitions p' finer than the starting partition can be refined in the same way and we will not prove this for each refining step.

nautyTree

If we look at how the cell 234678 is connected to 15 we see that 3 and 7 are the only two vertices that have no connection to 15 and we can therefore split the partition to [15|37|2468]. Now we look at how 2468 is connected to itself. The vertices 6 and 8 are connected to another vertex in 2468 but 2 and 4 are not. The partition can now be split further to [15|37|24|68]. Now we look at how 15 is connected to 24. 1 is connected to 24, but 5 is not. The partition can therefore be split to [15|37|24|68]. Finally we look at how 37 is connected to 24. 3 is connected to two vertices of 24 and 7 is connected to none. So we end get the partition [1|5|3|7|24|68].

Since this partition cannot be split further by refinement (it is not necessary to prove this, we would just be doing more work), the tree is split by case distinction of 24: we can color 24 so that $\gamma(2) < \gamma(4)$ or so that $\gamma(4) < \gamma(2)$ (where γ is the coloring).

In the left branch we have a case distinction again for the cell 68 and we get our first two end-nodes. The graphs represented by these end-nodes are isomorphic with isomorphism $(6, 8)$. The first leaf we get is [1|5|3|2|4|6|8] which is our reference partition p. If $p' = p$ we

get the identity. The second leaf is $p' = [1|5|3|7|2|4|8|6]$, which leads to the automorphism $(6, 8)$.

Now we return to the case $\gamma(4) < \gamma(2)$. Since we know that 6 and 8 are in the same orbit under permutations that stabilize 2 and 4 we can to assume $\gamma(6) < \gamma(8)$. This gives us another end-node $p' = [1|5|3|7|4|2|6|8]$, which gives another isomorphism with the first end-node: $(2, 4)$.

The automorphism group now becomes $\langle (2, 4), (6, 8) \rangle$. There are no automorphisms that interchange 1 and 5, and therefore the graphs are not isomorphic.

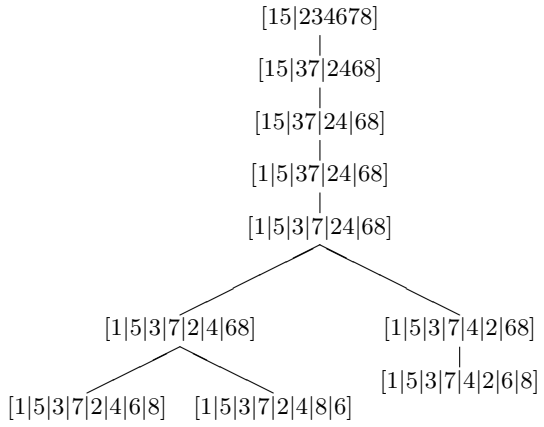

FIGURE 4. The search tree in McKay's algorithm

nautyTree

4. Towards a proof assistant

PAGN

We have developed a software package for automatically constructing a proof of (non)isomorphism of two given graphs. It is possible to ask for a specific proof by choosing invariants, calling Luks' algorithm, or calling McKay's algorithm. Conceivable the package be made more interactive by, for example, by using a vertex invariant as the coloring in the first step of MacKay's algorithm.

The software can derive the automorphism group of a single graph by calling the algorithm from Section 3. It can further derive a proof of graph nonisomorphism by using the graph automorphism algorithm from Section 3 in the following way.

mckay

mckay

The proof assistant and most of the algorithms assume that the graphs are connected. For example Luks's algorithm fails if the graphs are not connected. However it is relatively easy to reduce graph (non-) isomorphism of unconnected graphs to graph (non-) isomorphism of the connected components.

Because of the recursive nature of our proof, it is possible to modify the output for the level of mathematical sophistication of the user by removing low-level lemmas. The following example, using the graphs above, has been modified in this way.

```
Proposition: the graph G 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.
Proof:
  Suppose that p is an isomorphism that transforms G 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 can use information about the orbits of H under automorphisms 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 orbits.
  Proposition: The orbits of A are [ 1, 3 ] and [ 2, 4 ]. (proof hidden)
  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. (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 ].
  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:
      Look at [ 5:8 ]^pi and how it is connected to [5:8]^pi, for pi=p,p'.
      First for p
        [ 5:8 ]^p = [ 2 4 6 8 ].
        The vertices 2 4 are not connected to any vertices of [ 2 4 6 8 ].
        The vertices 6 8 are each connected to 1 vertex of [ 2 4 6 8 ].
      QED(p)
      Then for p'
        [ 5:8 ]^p' = [ 2 4 6 8 ].
        The vertices 2 4 are not connected to any vertices of [ 2 4 6 8 ].
        The vertices 6 8 are each connected to 1 vertex of [ 2 4 6 8 ].
      QED(p')
      If p^-1p' is an automorphism then it transfers [ 2 4 6 8 ] to [ 2 4 6 8 ] and must therefore
      transfer [ 2 4 ] to [ 2 4 ] and [ 6 8 ] to [ 6 8 ].

      Since [ 1 2 | 3 4 | 5 6 | 7 8 ]^p = [ 1 5 | 3 7 | 2 4 | 6 8 ], we now know that [ 1 2 | 3 4 | 5 6 |
      7 8 ]^p' = [ 1 5 | 3 7 | 2 4 | 6 8 ].
    QED(refine part)
    Lemma (refine part)
      If [ 1 2 | 3 4 | 5 6 | 7 8 ]^p' = [ 1 5 | 3 7 | 2 4 | 6 8 ] then p^-1p' does not interchange 1 and 5.
    Proof:
      Look at [ 7 8 ]^pi and how it is connected to [1 2]^pi, for pi=p,p'.
      First for p
        [ 7 8 ]^p = [ 6 8 ].
        [ 1 2 ]^p = [ 1 5 ].
        The vertex 1 is not connected to any vertices of [ 6 8 ].
        The vertex 5 is connected to 2 vertices of [ 6 8 ].
      QED(p)
      Then for p'
        [ 7 8 ]^p' = [ 6 8 ].
        [ 1 2 ]^p' = [ 1 5 ].
        The vertex 1 is not connected to any vertices of [ 6 8 ].
        The vertex 5 is connected to 2 vertices of [ 6 8 ].
      QED(p')
      If p^-1p' is an automorphism then it transfers [ 6 8 ] to [ 6 8 ] and [ 1 5 ] to [ 1 5 ] and must
      therefore transfer [ 1 ] to [ 1 ] and [ 5 ] to [ 5 ].

      Since [ 1 | 2 | 3 4 | 5 6 | 7 8 ]^p = [ 1 | 5 | 3 7 | 2 4 | 6 8 ], we now know that [ 1 | 2 | 3 4 |
      5 6 | 7 8 ]^p' = [ 1 | 5 | 3 7 | 2 4 | 6 8 ].
    QED(refine part)
    Lemma (refine part)
      If [ 1 | 2 | 3 4 | 5 6 | 7 8 ]^p' = [ 1 | 5 | 3 7 | 2 4 | 6 8 ] then p^-1p' does not interchange 1
      and 5.
    Proof:
      Look at [ 7 8 ]^pi and how it is connected to [3 4]^pi, for pi=p,p'.
      First for p
        [ 7 8 ]^p = [ 6 8 ].
        [ 3 4 ]^p = [ 3 7 ].
        The vertex 3 is not connected to any vertices of [ 6 8 ].
        The vertex 7 is connected to 2 vertices of [ 6 8 ].
      QED(p)
      Then for p'
        [ 7 8 ]^p' = [ 6 8 ].
        [ 3 4 ]^p' = [ 3 7 ].
        The vertex 3 is not connected to any vertices of [ 6 8 ].
        The vertex 7 is connected to 2 vertices of [ 6 8 ].
```

```
            QED(p')
            If p^-1p' is an automorphism then it transfers [ 6 8 ] to [ 6 8 ] and [ 3 7 ] to [ 3 7 ] and must
            therefore transfer [ 3 ] to [ 3 ] and [ 7 ] to [ 7 ].

            Since [ 1 | 2 | 3 | 4 | 5 6 | 7 8 ]^p = [ 1 | 5 | 3 | 7 | 2 4 | 6 8 ], we now know that [ 1 | 2 | 3
            | 4 | 5 6 | 7 8 ]^p' = [ 1 | 5 | 3 | 7 | 2 4 | 6 8 ].
          QED(refine part)
        QED(refinement)
        Now we look at all the different possibilities for [ 1 | 2 | 3 | 4 | 5 6 | 7 8 ]^p' = [ 1 | 5 | 3 | 7 |
        2 4 | 6 8 ] by looking at 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 ].
            Now we look at all the different possibilities for [ 1 | 2 | 3 | 4 | 5 | 6 | 7 8 ]^p' = [ 1 | 5 | 3
            | 7 | 2 | 4 | 6 8 ] by looking at 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.
                Further more it is included in H (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.
                Further more it is included in H (it is a generator of H).
              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 different possibilities for [ 1 | 2 | 3 | 4 | 5 | 6 | 7 8 ]^p' = [ 1 | 5 | 3
            | 7 | 4 | 2 | 6 8 ] by looking at 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).
              QED(case 7^p' = 6)
            QED(case distinction 7^p')
          QED(case 5^p' = 4)
        QED(case distinction 5^p')
      QED(automorphismgroup)
    QED(case v = 1)
    case v = 2

      From here on the proof is similar to the case v = 2 and hence deleted.

    QED(case v = 2)
  QED(case distinction)
QED(graphisomorphism)
```

The graphical frontend of our proof assistant 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 [gaplink 7]. From GAP a modified local copy of dreadnaut is called on demand. The information to dreadnaut is send in the format used by dreadnaut, the information sent back to GAP is sent in a simple XML format. For the link with GAP we use the OpenMath library [omlib 18] and GAP phrasebook from RIACA. They depend on the parsing library ANTLR [antlr 1].

References

antlr [1] *ANTLR Reference Manual*, 2005. Available from World Wide Web: `http://www.antlr.org/doc/index.html`.

BrouwerCohenNeumaier [2] A. E. Brouwer, A. M. Cohen, and A. Neumaier. *Distance-regular graphs*, volume 18 of *Ergebnisse der Mathematik und ihrer Grenzgebiete (3) [Results in Mathematics and Related Areas (3)]*. Springer-Verlag, Berlin, 1989.

Cohen [3] Arjeh Cohen and Scott Murray. Certifying solutions to permutation group problems. Lecture notes for the Calculemus Autumn School, Pisa, 23 Sep-4 Oct 2002. Available from World Wide Web: `http://www.win.tue.nl/~amc/pub/permgp.pdf`.

CohenAutomated [4] Arjeh Cohen, Scott Murray, Martin Pollet, and Volker Sorge. Certifying solutions to permutation group problems. In Franz Baader, editor, *19th International Conference on Automated Deduction*, pages 258–273. Springer-Verlag, Berlin, 2003.

Cormen1 [5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1990.

GAP4 [6] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2004. Available from World Wide Web: `http://www.gap-system.org`.

gaplink [7] RIACA GAP phrasebook, 2004. Available from World Wide Web: `http://www.mathdox.org/phrasebook/gap/`.

gxlcore [8] *GXL core 0.92 API*, 2004. Available from World Wide Web: `http://gxl.sourceforge.net/docs/gxl/api/index.html`.

gxl [9] Ric Holt, Andy Schrr, Susan Elliot Sim, and Andreas Winter. Graph exchange language 1.0 - dtd, 2001. Available from World Wide Web: `http://www.gupro.de/GXL/`.

jgraph [10] *JGraph v5.7.2 API Specification*, 2005. Available from World Wide Web: `http://www.jgraph.com/doc/jgraph/`.

jgrapht [11] *JGraphT: a free Java graph library*, 2005. Available from World Wide Web: `http://jgrapht.sourceforge.net/javadoc/`.

jlfgr [12] Java look and feel graph repository, 2000. Available from World Wide Web: `http://java.sun.com/developer/techDocs/hi/repository/`.

Kernighan88a [13] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Luks [14] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. System Sci.*, 25(1):42–65, 1982.

nauty [15] Brendan 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.

McKay77 [16] Brendan D. McKay. Computing automorphisms and canonical labellings of graphs. In *Combinatorial mathematics (Proc. Internat. Conf. Combinatorial Theory, Australian Nat. Univ., Canberra, 1977)*, volume 686 of *Lecture Notes in Math.*, pages 223–232. Springer, Berlin, 1978.

McKay81 [17] Brendan D. McKay. Practical graph isomorphism. In *Proceedings of the Tenth Manitoba Conference on Numerical Mathematics and Computing, Vol. I (Winnipeg, Man., 1980)*, volume 30, pages 45–87, 1981.

omlib [18] RIACA OpenMath library, 2004. Available from World Wide Web: `http://www.mathdox.org/omlib/`.

pagnurl [19] Proof assistant for graph non-isomorphism, 2006. Available from World Wide Web: `http://www.mathdox.org/graphiso/`.

mapplet [20] Erik Postma. RIACA mapplet, 2005. Available from World Wide Web: `http://www.mathdox.org/mapplet/`.

java [21] Sun. *Java 2 Platform, Standard Edition, v 1.4.2 API Specification*, 2004. Available from World Wide Web: `http://java.sun.com/j2se/1.4.2/docs/api/`.