Automatic proofs of graph nonisomorphism

Knopper, J.W.

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Automatic Proofs of Graph Nonisomorphism

By
J.W. Knopper

Supervisor: prof.dr. Arjeh M. Cohen
Supervisor: Dr. Scott H. Murray

Eindhoven, June 2006

Contents

Chapter 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. This proof is usually given in an encoded form, which is called a *certificate*. In this thesis, we construct such proofs for the problem of graph isomorphism.

A graph consists of a finite set of points, called *vertices*, and a set of *edges*, each of which connects two vertices. Two graphs are *isomorphic* if there is a (bijective) function that maps the vertices of the first graph to the vertices of the second with the property that there is an edge between two vertices, if and only if, there is an edge between their images. Such a function is called an *isomorphism*. Graph isomorphism is important because every finite combinatorial structure can be encoded in terms of graphs. For example, see [26, page 28] for an encoding of block designs.

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 ways 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. Because of this, it can take much longer to generate a proof than just to determine nonisomorphism. Although we are primarily interested in practical computations, we occasionally use the concept of polynomial time algorithms [7, Chapter 36].

In Chapter 2, we look at invariants: functions that take the same value on two isomorphic graphs, but may take different values on nonisomorphic graphs. In many cases, invariants give a short and easy-to-verify proof of nonisomorphism. For example, two graphs with different numbers of vertices 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. Graph isomorphism is hard in general, but there are several classes of graphs that can be handled in polynomial time [13, page 72]. In Chapter 3, we look at the algorithm of Luks [25], which takes polynomial time for classes of graphs with bounded vertex degree. In Chapter 4, we look at the algorithm of McKay [28], which is implemented as nauty [26]. This is fast in practice but is not known to be polynomial. We have modified both these algorithms to produce a human-readable proof. The modified version of McKay's algorithm can also produce a derivation of the correctness of the automorphism group of a graph.

We have developed a Proof Assistant for Graph Nonisomorphism [31]. 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 [9], apart from some modifications to nauty, which is in C [23]. The user interface is written in Java [35]. The proof assistant is described in Chapter 5.

Chapter 2

Invariants

2.1 Introduction

In this chapter, we discuss invariants that can be used to produce easy proofs of nonisomorphism for many pairs of graphs. Generally speaking, the more difficult an invariant is to compute, the more graphs it can distinguish. In fact there are *complete invariants*, which distinguish graphs perfectly, for example the canonical graph returned by nauty [26]. The invariants that are described in this chapter, however, are relatively easy to calculate and give short human-understandable proofs of nonisomorphism. They are also the invariants used in the proof assistant. Some of the invariants used are also found in [33].

2.2 Basic definitions

A *(simple) graph* $G = (V, E)$ consists of a finite set of vertices V and a set of edges E consisting of subsets of V with cardinality two. A *subgraph* of G is a graph $G' = (V', E')$, such that $V' \subseteq V$ and $E' \subseteq E$. The *induced subgraph* on a set of vertices $V' \subseteq V$ is the graph with vertices V' and edges $E' = \{\{v_1, v_2\} \in E \mid v_1, v_2 \in V'\}$.

Let $G = (V, E)$ and $G' = (V', E')$ be graphs. An *isomorphism* $\sigma : G \to G'$ is a bijective function from V to V' such that $E' = \{\{\sigma(v_1), \sigma(v_2)\} \mid \{v_1, v_2\} \in E\}$. An *automorphism* is an isomorphism from a graph G to itself. The *automorphism group* of a graph G is the group of all automorphisms of the graph and is denoted $\mathrm{Aut}(G)$. The *orbit* of a vertex under the automorphism group is the set of all images of that vertex by elements of the automorphism group.

Let $G = (V, E)$ be a graph and let $v_1, v_2 \in V$. Then v_1 and v_2 are called *adjacent* if $\{v_1, v_2\}$ is an edge. A *path* from v_0 to v_n is a sequence $(v_0, v_1, v_2, \ldots, v_n)$ of vertices such that v_{i-1} is adjacent to v_i, for $i = 1, \ldots, n$. The *length* of this path is n. If there is a path from every vertex to every other vertex, the graph is called *connected*. A *component* is a maximal connected induced subgraph. Note that a graph is connected if, and only if, it has exactly one component. If the n vertices of an induced subgraph are adjacent to every other vertex of that subgraph, the subgraph is called an *n-clique*. The *distance* from a vertex to another vertex is the minimum length of a path between those vertices, or ∞ if there is no such path. The *distance* between a vertex and an edge is the minimum of the distances to that vertex from each of the vertices of the edge. The *diameter* of a graph is the largest distance between two vertices. A *cycle* is a path $(v_0, \ldots, v_n)$ with $n > 1$, where $v_i = v_j$ if, and only if, $\{i, j\} = \{0, n\}$ or $i = j$. The *girth* of a graph is the length of the shortest cycle or ∞ if the graph has no cycles. The *degree* (or *valence*) of a vertex is the number of edges containing that vertex.

Choose some order on the elements of V, say $V = \{v_1, \ldots, v_n\}$. The *adjacency matrix* A is the $n \times n$ matrix with the value 1 at position (i, j) if v_i and v_j are adjacent, and the value 0 otherwise. Note that A can be written over any *unital ring*, that is a ring with an *unit*, which is an invertible element. The *distance matrix* is the matrix of size $n \times n$ where the value at position

(i, j) is the distance between v_i and v_j. This matrix can be calculated in $O(|V|^3)$ time using the Floyd-Warshall Algorithm [7, page 560].

A *tree* is a connected graph $G = (V, E)$ with no cycles. A *rooted tree* is a tree with a distinguished vertex called the *root*. A vertex in a tree is also called a *node*. Note that in a tree there is a unique shortest path from each node to the root. Let v be a node in the tree other than the root and $e = (v', v)$ be the first edge on the path from v to the root. We then call v' the *parent* of v and v the *child* of v'. A node with children is called an *internal node*, while a node without children is called an *external node* or *leaf*.

There are several standard computer representations of graphs. The ones we use are the list of vertices and edges, by the adjacency matrix (the definition can be found below) or by the *edge list* representation: a graph is represented by the set of vertices and for each vertex the set of adjacent vertices. We can convert between the different representations in time at most $O(|V|^2)$. Since nearly all the algorithms we consider need more time than this, we do not consider the choice of representation to be a significant factor. More about representations and the O notation can be found in [13, Section 2.1].

2.3 Graph Invariants

A *graph invariant* is a function on the class of all graphs that takes identical values on isomorphic graphs. In order to prove that two graphs are nonisomorphic, it suffices to find a graph invariant that has different values for these graphs. There are graph invariants that give nice short proofs of graph nonisomorphism.

Some graph invariants always take different values on nonisomorphic graphs (e.g., the lexicographically-least adjacency matrix [24, page 253] or the canonical label of [28, Section 2.1, 2.17]), but they are very hard to compute and lead to long proofs.

Number of vertices

This invariant takes as value the number of vertices of the graph.

Number of edges

This invariant takes as value the number of edges of the graph.

Number of triangles

This invariant takes as value the number of *triangles* in the graph, i.e., sets of vertices $\{u, v, w\}$ such that $\{u, v\}, \{u, w\}, \{v, w\} \in E$. This can be computed in time $O(|E||V|)$. Note that the number of vertices and the number of edges are the number of 1-cliques and 2-cliques respectively. This algorithm can be extended to give the number of m-cliques. Cliques can be found uniquely by introducing a total order on the vertices and selecting the smaller vertices first. This leads to a worst case complexity $O(|E||V|^{m-2})$ for $m \geq 2$.

Besides looking at the number of m-cliques we can also look at the number of vertices connected to an $(m-1)$-clique and the number of edges between those vertices.

Diameter

This invariant takes as value the diameter of the graph. First the distance matrix is calculated using the Floyd-Warshall Algorithm [7, page 560]. The diameter is then the maximum of the entries. If the graph is disconnected, the diameter is infinite, but we can instead use the multiset of diameters of the components.

Girth

This invariant takes as value the girth of the graph. The girth can be calculated using a breadth-first search [7, Section 23.2] for each vertex, as in Algorithm 1. For each vertex v we try to find two different paths to another vertex w, which minimum total length. This is not necessarily a cycle, but this does contain a cycle, with length less or equal than the combined length of the paths.

For each cycle, on which v lies there is a vertex w such that there are two different paths from v to w, of which the lengths differ at most one. Therefore we can restrict ourselves to the case where the lengths of the paths differ only one.

In each search, the vertices are first colored white (at distance *dist* or further), then they become gray (light gray at distance *dist* and dark gray at distance $dist - 1$), and eventually black (at distance smaller than *dist*). Initially one vertex is light gray and the rest are white. In each step we color the dark gray vertices black and the light gray vertices dark gray and look at the vertices connected to them.

If a white vertex is encountered it is colored light gray. Since it is the first path to it, nothing further is done. If a vertex that is light gray or dark gray is encountered, we have found a cycle of length at most $2 * dist$ or $2 * dist - 1$.

This algorithm can be extended to return a cycle of smallest length. To prove the correctness of the given value for the girth, it is enough to prove that no smaller cycle exists. So it is enough to initialize the girth with the given value instead of ∞. The algorithm takes time $O(|V||E|)$: the outer loop is run $|V|$ times, and each edge is checked at most twice in the inner while loop.

Polynomial invariants

The *characteristic polynomial* of a graph is the determinant $P_G(\lambda) = |\lambda I - A|$, where A is the adjacency matrix and λ is an indeterminate. The characteristic polynomial can be computed with $O(|V|^3)$ ring multiplications. In order to control the time taken for a ring multiplication, we often use the ring $\mathbb{Z}/m\mathbb{Z}$, for some m.

There are several other interesting *polynomial invariants*. Let $n = |V|$ and let J be the $n \times n$ all-one matrix. Define the *Seidel matrix* $S = J - I - 2A$. Let d_i be the degree of v_i. The *degree matrix* D is the $n \times n$ diagonal matrix with the value d_i at position (i, i) and the value 0 elsewhere. Assume that every $d_i > 0$, so that $|D| > 0$. The following characteristic polynomials are now also invariants:

$$Q_G(\lambda) = \frac{1}{|D|}|\lambda D - A|, R_G(\lambda) = |\lambda I - D - A|,$$
$$S_G(\lambda) = |\lambda I - S|, \text{ and } C_G(\lambda) = |\lambda I - D + A].$$

If the graph is *vertex regular*, that is all vertices are in the same orbit under the group of automorphisms of the graph, then some of these polynomials contain the same information [36].

The *roots* of a characteristic polynomial are the eigenvalues of the adjacency matrix. Because the adjacency matrix is symmetric, that is $A = A^T$, these eigenvalues are real. The set is also called the *spectrum* of a graph. More information about spectra can be found in [8] and [36].

The characteristic polynomial of the distance matrix is also an invariant [8, Section 9.2].

Smith normal form

Let R be a principal ideal domain [6]. Every $n \times n$ matrix A over R can be transformed with elementary operations on rows and columns to a diagonal matrix D with the property that $D_{i,i} \mid D_{i+1,i+1}$, for all $0 < i < n$. In other words, there exist invertible P and Q such that $PAQ = D$. For $R = \mathbb{Z}/p\mathbb{Z}$, where p is a prime, it is relatively easy to compute the Smith normal form. See also [6].

The Smith normal form of the adjacency matrix is an invariant up to units. This is comparable with essentially unique factorization in $\mathbb{Z}$, where -1 is a unit and $4 = 2 \cdot 2 = (-2) \cdot (-2)$. To make this an invariant we have to perform the analogue of taking the positive number. If R is a field, then computing the Smith normal form is equivalent to computing the rank of A.

Algorithm 1 Calculating the girth

Input: $G = (V, E)$ is a graph
Returns: the *girth* of G

```
1: function GIRTH(G)
2:     var
3:         girth:integer                                   ▷ upper bound for the girth of G
4:         u:vertex                                                              ▷ u ∈ V
5:         v:vertex                                                              ▷ v ∈ V
6:         w:vertex                                                              ▷ w ∈ V
7:         dist:integer                                    ▷ related to the distance to v
8:         a list of LIGHT GRAY vertices                   ▷ at distance dist of v
9:         a list of DARK GRAY vertices                    ▷ at distance dist − 1 of v
10:        a list that contains for each vertex the color  ▷ to check whether a vertex is WHITE
11:    end var
12:    girth := ∞
13:    for v in V do
14:        color v LIGHT GRAY and all other vertices WHITE.
15:        dist := 0
16:        while the are LIGHT GRAY vertices do
17:            color all DARK GRAY vertices BLACK.
18:            color all LIGHT GRAY vertices DARK GRAY.
19:            dist := dist + 1   ▷ now all DARK GRAY vertices are at distance dist − 1 of v and
                   there are no LIGHT GRAY VERTICES
20:            if girth > 2 ∗ dist then                    ▷ improving girth might be possible
21:                for DARK GREY vertices u do
22:                    for w adjacent to u do
23:                        if w is colored WHITE then      ▷ w was not reached before
24:                            color w LIGHT GRAY
25:                        else if w is LIGHT GRAY then    ▷ we now have two different paths of
                               length dist to w
26:                            if girth > 2 ∗ dist then
27:                                girth := 2 ∗ dist       ▷ w and u are both at distance dist from v,
                                       therefore we have found a cycle of size ≤ 2 ∗ dist.
28:                            end if
29:                        else if w is DARK GRAY then     ▷ we now have two different paths of
                               lengths dist − 1 and dist to w
30:                            if girth > 2 ∗ dist − 1 then
31:                                girth := 2 ∗ dist − 1   ▷ w is at distance dist − 1 and u is
                                       at distance dist from v, therefore we have found a cycle of size
                                       ≤ 2 ∗ dist − 1.
32:                            end if
33:                        end if
34:                    end for
35:                end for
36:            end if
37:        end while
38:    end for
39:    return girth
40: end function
```

2.4 Vertex invariants

A *vertex invariant* assigns to every graph G a function f_G on the vertex set of G, such that $f_G(v) = f_{G'}(\sigma(v))$, whenever v is a vertex of G and $\sigma : G \to G'$ is an isomorphism. In this section, we describe some easy-to-calculate vertex invariants. They can be transformed into graph invariants as follows: Given a graph $G = (V, E)$ and vertex invariant f_G, the multiset $f_G(V) := \{f_G(v) \mid v \in V\}$ is a graph invariant.

Degree

The degree of a vertex v is the number of vertices $u \in V$ for which $\{u, v\} \in E$. It takes time $O(|V| + |E|)$ to calculate this for all vertices.

Distance multiplicities

The distances between all pairs of vertices can be calculated with the Floyd-Warshall algorithm, which gives the distance matrix. For each vertex, the multiset of distances to all other vertices is a vertex invariant.

Number of triangles

The number of triangles containing the vertex is a vertex invariant, which can be calculated in time $O(|E|)$ per vertex.

Subgraph invariance

Consider the subgraph induced on the set of vertices at distance one from a vertex. Invariants of this subgraph are also invariants of the vertex. In particular, we use the multiset of the sizes of the components of this subgraph as a vertex invariant in the proof assistant. Once the distance matrix is known, this invariant can be calculated in time $O(|V| + |E|)$ for each vertex. Taking the union of this invariant (instead of the multiset) we get a somewhat less powerful, but shorter graph invariant.

Extended Subgraph Invariance

The isomorphism class of the subgraph induced on all vertices at distance (at most) i from a vertex is also a vertex invariant. As with subgraph invariance we use the sizes of the components as an invariant. When the distance matrix has been calculated this can be calculated in time $O(|V| + |E|)$ for each vertex.

Powers of the Adjacency Matrix

The set of powers of the adjacency matrix is not itself an invariant, but can easily be turned into one. The kth power of the adjacency matrix A has at position (i, j) the number of paths of length k from v_i to vertex v_j. Since the adjacency matrix is a solution to the characteristic polynomial, which has degree n, it is enough to look at the first $n - 1$ powers of the adjacency matrix. These powers can be calculated in time $O(|V|^4)$.

We now define the multisets

$$M_i := \{(A_{ij}, {A^2}_{ij}, \ldots, {A^{n-1}}_{ij}) \mid j = 1, \ldots, n\}$$

It is easily seen that the map $f_g : v_i \to M_i$ is a vertex invariant. In computation, the multiset is stored as a list, sorted with respect to the lexicographic order. The sorting takes time $O(|V|^4)$, so this is the time required to calculate the invariant.

2.5 Edge invariants

An *edge invariant* assigns to every graph G a function f_G on the set of edges of G, such that $f_G(e) = f_{G'}(\sigma(e))$ whenever e is an edge of G and $\sigma : G \to G'$ is an isomorphism. Vertex invariants can be transformed into graph invariants as follows: Given a graph $G = (V, E)$ and vertex invariant f_G, the set $g_G(\{v_1, v_2\}) = \{f_G(v_1), f_G(v_2)\}$ is an edge invariant. Edge invariants can be transformed into graph invariants as follows: Given a graph $G = (V, E)$ and edge invariant f_G, the multiset $f_G(D) := \{f_G(v) \mid e \in E\}$ is a graph invariant.

Number of triangles

The number of triangles containing the edge is an edge invariant.

Number of $K_{2,1,1}$ graphs

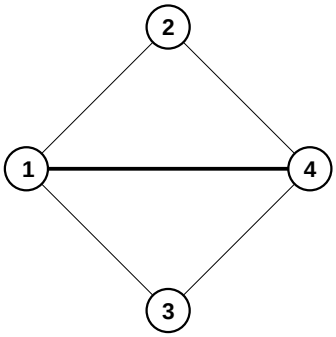

Figure 2.1: The $K_{2,1,1}$ graph, with the edge e bold

The number of $K_{2,1,1}$ graphs around an edge is an edge invariant, see Figure 2.1. This can be generalized to looking at the number of vertices connected to both vertices of an edge and the number of edges between them.

It can be further generalized to the number of vertices connected to all vertices in an m-clique and the number of edges between them. This can be made an edge invariant if we only count cliques that include a given edge. This generalization is harder to compute than the generalization as a graph invariant in Section 2.3 because we count each clique once for each edge in it.

Edge distance multiplicities

This invariant is comparable to the distance multiplicity vertex invariant. For an edge $e = \{v_1, v_2\}$, we look at the distances of all other vertices from v_1 and v_2. The distance of a vertex is either the same to both v_1 or v_2, or differs by one. We can therefore count the vertices of the graph according to their distance to e and whether they are closer to v_1 or v_2, or have the same distance to both. The result consists of three lists of numbers of vertices. The list l_1 of vertices closest to v_1, the list l_e of vertices with equal distance, and the list l_2 of vertices closest to v_2. The invariant is now $(\min(l_1, l_2), l_e, \max(l_1, l_2))$, under the lexicographical ordering.

Below is an example of the value of this invariant on an edge in a graph on 8 vertices. The numbers at the top represent vertices closer to v_1 and the numbers at the bottom represent vertices closer to v_2. The numbers in the middle represent vertices with the same distance to both v_1 and v_2. Here v_1 and v_2 are chosen so that the top row is lexicographically not larger than the bottom row.

```
the (sorted) distance multiplicity of the edge is:
. 1 0
|2 1
. 2 0
```

If the distance matrix is known this can be calculated in $O(V)$ time per edge.

2.6 Invariants in the Proof Assistant

To check whether two graphs are non isomorphic, the functions in the array `AllGraphInvariants` are checked in the order they are in that array. At the moment this array consists of the following functions. They are chosen in a way that the invariants that are easy to humanly understand or easier to calculate are checked earlier. In larger graphs some of the invariants high in the tree become harder to humanly verify, but still can give information about the graph.

Invariants like the number of vertices and the number of edges are checked first. Invariants that take more time to compute and are harder to humanly verify like characteristic polynomials and the Smith normal form are in the middle. And at the end are some invariants, which are often useful to distinct graphs, but do not always give useful information and are harder to verify, like the number of $K_{2,1,1}$ graphs per edge and the powers of the adjacency matrix.

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
10. characteristic polynomial (type Q)
11. characteristic polynomial (type R)
12. characteristic polynomial (type S)
13. Smith normal form
14. powers of the adjacency matrix
15. number of triangles per vertex, edge (multiset)
16. number of $K_{2,1,1}$-graphs per edge (multiset)
17. edge distance multiplicity
18. multiset of all edge invariants per edge

Note that some invariants are straightforward to calculate but harder to prove. In such a case it is easier to calculate the value than to check a lengthy implemented. 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.

2.7 Colored graphs

We often find it convenient to expand our definition of a graph by allowing vertices of different types. In this case we only consider isomorphisms that transform vertices to other vertices of the same type. The types are called *colors*, and the resulting structure is a *colored graph*. Nauty (Chapter 4) already supports colored graph isomorphism, and we have extended Luks' algorithm (Chapter 3) so that it also supports colored graph isomorphism.

A *(vertex) coloring* of a graph is a function γ from the set of vertices to a fixed set of colors C. Let $G = (V, E, \gamma)$ and $G' = (V', E', \gamma')$ be colored graphs. Let σ be an isomorphism from (V, E) to (V', E'). We say that σ is an *isomorphism* from G to G' if the colors are preserved, that is, $\gamma(v) = \gamma'(\sigma(v))$ for every $v \in V$. If C is totally ordered and γ is onto, we refer to γ as a *partition*.

In this section, we describe how color can be used to improve invariants, and how to combine colorings to make a finer coloring.

2.7.1 Colors and graph isomorphism

Coloring can be useful in graph isomorphism because it can reduce the number of possibilities and therefore speed up the algorithm. Some coloring may come from the initial problem and additional coloring may come from vertex invariants since each of them is a coloring (see the next section). Furthermore it is sometimes possible to make a coloring *finer*, that is containing more information about which vertices are different. An example is refining, which is described in Chapter 4.

2.7.2 Colors and invariants

Note that a vertex invariant f gives a function f_G from the vertices of G to some set C. Each of these maps can be considered as a coloring.

The fact that f is a vertex invariant is equivalent to saying that every graph isomorphism $\sigma : G \to G'$ is also a colored graph isomorphism $(G, f_G) \to (G', f_{G'})$.

In the calculation of some invariants the value of a subfunction on vertices (or edges) is sorted. It is possible to take coloring into account here, which can result in a better invariant. An example here are graph invariants from vertex invariants, like the degree multiset.

Chapter 3

Luks' algorithm

3.1 Introduction

The graph isomorphism problem is hard in general, but it can be solved in polynomial time for certain classes of graphs. An example is the class of graphs of degree at most c, for some fixed constant c. In this chapter, we describe an algorithm due to Luks [25]. This algorithm calculates the group of all automorphisms of a graph that leave a given edge fixed. In Section 5.1 we show how to use this to solve the isomorphism problem. For graphs of degree $\leq c$, with n vertices, Luks' algorithm takes time $O(n^{5c})$. Our description of the algorithm is taken from [17]. In Section 3.6, we discuss how we adapted the algorithm for colored graphs.

3.2 Definitions

Let G be a (permutation) group and X a set. A group action of G on X is a mapping $\mu : X \times G \to X$, such that for all $g, h \in G$, and $x \in X : \mu(x, gh) = \mu(\mu(x, g), h)$ and $\mu(x, 1) = x$. We also say that G *acts* on X by μ. If it is clear which action is meant we say G acts on X and write $\mu(x, g) = x^g$, for $x \in X$ and $g \in G$.

Let G be a (permutation) group that acts on a set X. Let $x \in X$. The *stabilizer* of x by G is the subgroup $\{g \in G \mid x^g = x\}$ of G. Let $Y \subseteq X$. The *setwise stabilizer* of Y by G is the subgroup $\{g \in G \mid \forall_{y \in Y} y^g \in Y\}$. The *pointwise stabilizer* (or *vertexwise stabilizer*) of T by Y is the subgroup $\{g \in G \mid \forall_{y \in Y} y^g = y\}$.

Let X be a set and G a group that acts on X. Then the *orbit* of $x \in X$ under G is the set of the images of x under the action of G: $\{x^g \mid g \in G\}$.

Let G be a (permutation) group. We say that G has *generating set* $A = \{a_1, \ldots, a_k\}$ if each element $g \in G$ can be written as a finite product of the a_i. We also write $G = \langle A \rangle$. The representation used for a groups is the set of its generators. The group generated by $A' = A \cup \{a_{k+1}\}$ can be written as $\langle A' \rangle = \langle A \cup \{a_{k+1}\} \rangle$ but also as $\langle G \cup \{a_{k+1}\} \rangle$.

Let $G = (V, E)$ be a connected graph and let $e = \{v_1, v_2\} \in E$ be a fixed edge. Let $\mathrm{Aut}_e(G)$ be the group of automorphisms of G which fix e setwise. Let c be an upper bound on the degree of G. Define V_i to be the set of vertices at distance i from e. Define E_i to be the set of edges with one vertex in V_i and the other in V_i or V_{i+1}. Note that V is the disjoint union of the V_i and E is the disjoint union of the E_i. Define G_i to be the subgraph $(V_0 \cup \ldots \cup V_i, E_0 \cup \cdots \cup E_{i-1})$. Let h be the largest value of i, for which V_i is non-empty. Note that the sets E_i are nonempty for $0 \leq i \leq h-1$ and that E_h can be empty. Furthermore $G_{h+1} = G$. Let $A_i = \mathrm{Aut}_e(G_i)$. Clearly $V_1, \ldots, V_h$ are nonempty and V_i is empty for $i > h$.

The *ancestry* of $v \in V_k$ is the set of vertices $u \in V_{k-1}$ such that $\{u, v\} \in E_{k-1}$. If $\{u, v\}$ is an edge with $u \in V_{k-1}$ and $v \in V_k$, then we say that the edge has *type* (i, j) where i is the cardinality of the ancestry of v, and j is the number of vertices in V_k with the same ancestry as v, including

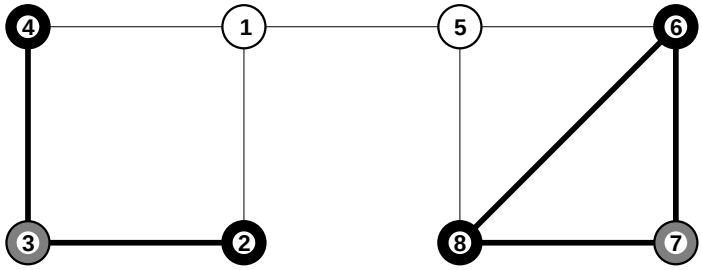

Figure 3.1: An example of splitting a graph with fixed edge $\{1,5\}$ in $V_0 = \{1,5\}, V_1 = \{2,4,6,8\}$, and $V_2 = \{3,7\}$. E_0 contains the thin edges and E_1 the bold edges. E_2 is empty.

v itself. If $\{u,v\}$ is an edge with u and v in V_{k-1} for some k, we say that the edge is of *type* $(2,0)$. Note that every edge has a type and the number of different types that occur is at most c^2.

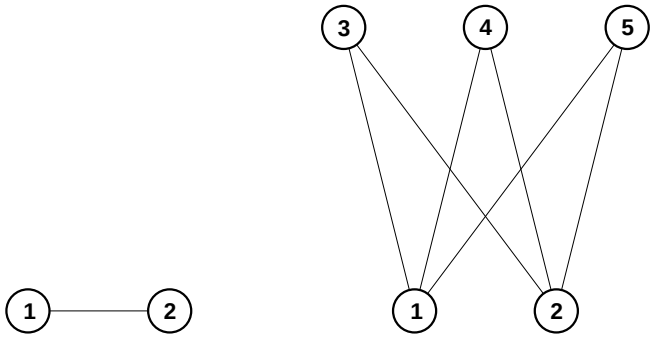

Figure 3.2: A family of type $(2,0)$ (left) and one of type $(2,3)$ (right).

Let f be an edge of type other than $(2,0)$. Suppose that $f = \{u,v\}$ with $u \in V_{k-1}$ and $v \in V_k$. The *family* of f consists of the set of edges connecting a vertex in the ancestry of v with a vertex in V_k with the same ancestry. The *family* of an edge of type $(2,0)$ is the set containing only that edge. If F is a family of edges of type (i,j), then F_{k-1} denotes the set of i corresponding vertices in V_{k-1} and F_k denotes the set of j corresponding vertices in V_k. If F is a family of edges of type (i,j), then F is also said to be of type (i,j). With $\mathcal{F}$ we denote the list, that for each type contains, the set of all families of edges in E_{k-1} with that type.

An example is given in Figure 3.2. The vertices 1 and 2 are in V_{k-1}, while the vertices 3, 4, and 5 are in V_k. The family on the left is denoted by $(F_{k-1} = \{1,2\}, F_k = \emptyset)$ and the family on the right by $(F_{k-1} = \{1,2\}, F_k = \{3,4,5\})$. Note that a family specifies a complete bipartite subgraph.

3.3 Permutation algorithms

In Luks' algorithm, several permutation group algorithms are used. In order to produce a proof of correctness for Luks' algorithm, we must also produce proofs for these algorithms. We use the methods of Cohen and Murray [3] to give human readable proofs. These are briefly explained in this section. Some extensions were needed to adapt them for use in Luks' algorithm, as described in Section 3.3.3. A list of all the subprocedures used to give proof can be found in Section 3.3.4. More about generating human readable proofs can be found in Section 3.3.3.

```
Proposition: Let H be the group generated by h1 = (1,2,3) and h2 = (2,3,4).
Let G be the group generated by g1 = (1,2,3,4) and g2 = (1,2). Then H is a
subgroup of G.
Proof:
  It suffices to verify that each of the generators of H belongs to G. Below
  we give a word in the generators of G for each generator of H.
  Proof:
    h1 = (1,2,3) = g1^2*g2*g1^3;
    h2 = (2,3,4) = g1*g2.
  QED(proof by generator)
  This establishes that each generator of H belongs to G, and so H is a
  subgroup of G.
QED(issubgroup)
```

Figure 3.3: An example of a subgroup proof

3.3.1 Membership and subgroup testing

To prove that a permutation is a member of a permutation group, it is sufficient to write it as a product of the generators of the group. To prove that a group H is a subgroup of a group G, it suffices to prove that all generators of H are members of G. There are standard algorithms in GAP to write permutations as words in generators. Figure 3.3 shows the output of the proof that the alternating group on 4 symbols $H = \langle (1,2,3), (2,3,4) \rangle$ is a subgroup of the symmetric group on 4 symbols $G = \langle (1,2,3,4), (1,2) \rangle$.

3.3.2 Schreier trees

A *directed graph* $G = (V, E)$ consists of a finite set of vertices V and a set of edges E consisting of pairs (v_1, v_2) of vertices $v_1, v_2 \in V$, $v_1 \neq v_2$. Note that the difference with a simple, or undirected, graph is that these edges are ordered: if $(v_1, v_2) \in E$ then we write $v_1 \to v_2$. For a directed graph $G = (V, E)$ the *corresponding simple graph* is the graph $G' = (V, E')$, with $E' = \{\{v_1, v_2\} \mid (v_1, v_2) \in E\}$. Let C be a set and $\gamma : E \to C$ a function. Then γ is called an *edge coloring*. A *spanning tree* of a connected graph is a connected subgraph, that contains all vertices and no cycles. A spanning tree can be formed from a connected graph by removing an edge from a cycle until no more cycles exist. A *directed rooted tree* is a directed graph $G = (V, E)$ with root $x \in V$ of which the edge between a non-root vertex v and its parent v' in the corresponding simple graph G_s (v', v) is in E but (v, v') is not. Note that in G_s the distance of $d(v, x) = d(v', x) + 1$ and that if there exists for two connected vertices $v, w \in V : v \neq w \wedge d(v, x) = d(w, x)$ there must also exist a cycle, which is a contradiction and therefore this construction holds.

Let X be a set, $x \in X$, G a group that acts on X and $A = \{g_1, \ldots, g_n\}$ the generating set of G. As defined earlier the orbit of x under G is the set $x^G = \{x^g | g \in G\}$. Now let $V = xG$ and $E = \{y \to z \mid y_i^g = z\}$. Let γ be a coloring from E to $C = \{1, \ldots, n\}$, so that $\gamma(\{y \to z\})$ is the index of a generator that maps y to z. We also write $y \xrightarrow{i} z$. Then the *orbit graph* is the directed graph $\mathcal{G} = (V, E)$ with edge coloring γ, see Figure 3.4 for an example. Consider the the set $A' = \{g_1, \ldots, g_n, g_n^{-1}, \ldots, g_1^{-1}\} = \{g'_1, \ldots g'_{2n}\}$. This set contains the generating set of G and all its elements are elements of G and therefore this set is also a generating set of G. Furthermore we now have that if $y \xrightarrow{i} z$ then $z \xrightarrow{2n+1-i} y$, meaning that we can choose the directions of the arrows as we like. Let S be a spanning tree constructed from the corresponding simple graph. It is now possible to create from S a rooted directed graph $\mathcal{S}$ with root α by choosing directions for the edges. It is furthermore possible to create an edge coloring $\gamma : E \to \{1, \ldots, 2n\}$. Then $\mathcal{S}$ is called a *Schreier Tree* with root α for the generating set A.

To generate a Schreier Tree it is not needed to construct it in the way mentioned above. An easier construction is to start with x and for all vertices in the orbit of x to calculate for all generators $a \in A$ whether x^a or $x^{a^{-1}}$ is a vertex not yet in the orbit. If it is not yet in the orbit

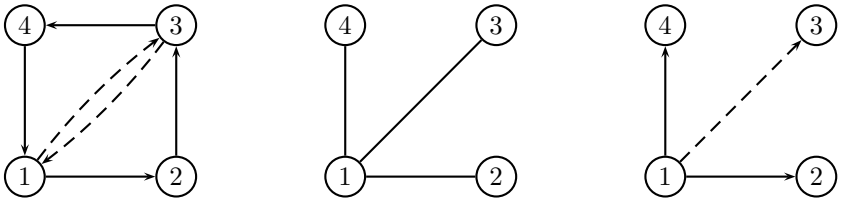

Figure 3.4: For the group $\langle (1,2,3,4),(1,3) \rangle$ the (directed) orbit tree, a spanning tree S and a Schreier tree $\mathcal{S}$

then it should be added with the edge corresponding to a or a^{-1}. The precise algorithm can be found in [3, Algorithm 5.1].

Let G be a group that acts on a set X. Let $\alpha \in X$. Let $\mathcal{S}$ be a Schreier Tree with root $\alpha \in X$. Let ω be an element of the orbit α^G. Then ω is a node in the $\mathcal{S}$. Consider the shortest path from α to ω. If the generators corresponding to the edges are multiplied in the order the arrows are encountered, a permutation is obtained that transfers α in ω. Define the *Schreier element* t_ω of ω as that permutation. Now let $p \in A'$ and $\beta \in \alpha^G$. Then $\alpha^{t_\beta p t_{\beta^p}^{-1}} = \beta^{p t_{\beta^p}^{-1}} = \alpha$. Or for all β the permutation $\alpha^{t_\beta p t_{\beta^p}^{-1}}$ stabilizes α. This leads to Schreier's lemma:

Lemma 3.1 (Schreier's lemma)**.** *Let G be a group with generating set A, acting on a set X, with $\alpha \in X$. Let $\mathcal{S}$ be a Schreier Tree with root α for A. Then the stabilizer of α by G is generated by the set of Schreier Elements:*

$$\left\{ \alpha^{t_\beta p t_{\beta^p}^{-1}} \mid \beta \in \alpha^G, p \in A' \right\}.$$

For a proof of Schreiers lemma see [5, Lemma 2.15] A large part of the calculations are spent on Stabilizer calculations. For more information about Schreier Trees see [5, Section 8.2] and [34, Chapter 4]. To pointwise stabilize multiple points it is best to stabilize them one at the time.

3.3.3 Extensions

Actions

The original algorithms from [3] are designed for permutation groups acting on integers. This is not always the case in Luks' algorithm. We want to find the permutations that stabilize the list of sets of families, by stabilizing the sets of families one by one.

In GAP [9] this can be solved by using *actions*. There are some actions predefined in gap, for example `OnSets` on sets. Where the action needed did not already exist we have created our own. To generate proofs in this situation, the original algorithms were modified, so that they now support actions.

We illustrate actions by a small example. Let G be the graph from Figure 3.5. Suppose we want to calculate and get a proof of the orbit of the set $\{1,3\}$ under the group of automorphisms of G. It is now possible to do this. The output of the extended algorithm is shown in Figure 3.6.

Generating human readable proofs

The structure used is based on the methods from [3]. Here the more complex proofs are made using smaller lemmas and their proofs, generated by other methods. To proof Luks' algorithm we use several of these methods and also have created several more smaller methods. For more information about which proving methods exactly are used and which ones are created see 3.3.4.

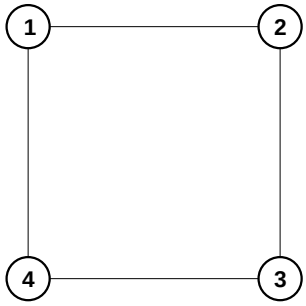

Figure 3.5: A simple graph on four vertices

```
Proposition: the orbit of x = [ 1, 3 ] under A, the group generated by a1 =
(2,4) and a2 = (1,2)(3,4), is X = [ [ 1, 3 ], [ 2, 4 ] ].
Proof:
  By the straightforward check that the cycles containing points of X do not
  contain points not in X, it follows that each generator of A leaves X
  invariant. It remains to show that the points in X are images of x under
  elements of A:
    [ 1, 3 ] = [ 1, 3 ]^()
    [ 2, 4 ] = [ 1, 3 ]^(1,2)(3,4) = [ 1, 3 ]^a2
QED(orbit of x = [ 1, 3 ])
```

Figure 3.6: The orbit of $\{1, 3\}$ in the graph from Figure 3.5

Since a lot of lemmas have their own lemmas it is useful to display the proof so that the structure is clear. In the original methods the beginning end ending of a method were clearly visible, but it was harder to get a greater structure. The commands that output text have been rewritten to use newly created functions from `graphiso.gi`, to allow more flexibility. Most functions now print a statement, and then the proof, so that it becomes possible to only show the statement.

It is now possible to generate formatted text output, HTML output and a private XML format. The formatted text has an indentation of 2 and a line width that can be set. In the HTML output, the proofs are implemented as lists, which can be expanded on click by use of pureDOMexplorer [16]. The proofs included in this thesis are generated as text with line width 159.

Since some proofs are long it might be desirable to not provide them unless the user asks for them. This can the proof shorter. In the short version of the proof, the orbit and stabilizer proofs are not shown (but the proof statements, or lemmas are).

Other output: HTML/XML

The original algorithms from [3] gave text output. While this is sufficient for small proofs, it is desirable to have more structure in a long proof. The algorithms have been adapted to produce HTML or XML.

Pointwise stabilizers

The Schreier-Sims algorithm can easily be adapted to calculate the pointwise stabilizer of some subset (of V)[34, 5.1.1].

3.3.4 List of all algorithms

The subprocedures that give a proof and their function are mentioned here. The file `permgps.gi` is the while where the (modified) original algorithms from [3] can be found and the file `luks.gi` is the file where the other functions can be found.

The (modified) algorithms in `permgps.gi` are:

- VerbalIsBase_Proof$(G, B, Gletter, truncatej)$ gives a proof that B is a base for G with elements $truncatej$ (if $truncatej = 0$ then the elements will be calculated). This function is not used.
- VerbalIsIn_Proof$(g, G, gletter, Gletter)$ gives a proof that $g \in G$.
- VerbalIsNotIn_Proof$(g, G, gletter, Gletter)$ gives a proof that $g \notin G$. This function is not used.
- VerbalIsSubgroup_Proof$(H, G, Hletter, Gletter)$ gives a proof that H is a subgroup of G
- VerbalIsPointwiseStabilizer_Proof$(G, vset, H, Gletter, Hletter[, action])$ gives a proof that H is the pointwise stabilizer of $vset$ by G under the action $action$ (default is $OnPoints$). The points are stabilized in the order that they occur in $vset$. This function does not give output if stabilizer-proofs are hidden. This function is new.
- VerbalIsStabilizer_Proof$(G, x, H, Gletter, Hletter[, action])$ gives a proof that H is the stabilizer of x by G under the action $action$ (default is $OnPoints$). Note that is not an efficient way to calculate the pointwise stabilizer of a set. The function VerbalIsPointWiseStabilizer_Proof was newly created for that purpose. This function does not give output if stabilizer-proofs are hidden.
- VerbalOrbit_Proof$(G, x[, action])$ gives a proof of the orbit of x under G where the action is $action$ (default is $OnPoints$). This function does not give output if orbit-proofs are hidden.
- VerbalOrder_Proof(G) gives a proof of the order of G. This function is not used.
- VerbalSchreierData$(G, x, Gletter[, action])$ computes the Schreier data of x in G under the action $action$ (default is $OnPoints$).

The algorithms in `luks.gi` are:

- VerbalAutomorphism_Proof : see Algorithm 3
- VerbalFamilyBlocks_Proof : see Algorithm 4
- VerbalGetFamilyPerm_Proof : see Algorithm 6

3.4 Luks' algorithm

Here we describe Algorithm 2, an outline of Luks' algorithm. First all the sets V_i and E_i are calculated in the straightforward manner. Clearly A_0 is generated by the permutation that swaps the vertices of e and fixes all other vertices. The algorithm now inductively constructs A_k from A_{k-1} for $k = 1, \ldots, h+1$ and then returns $A_{h+1} = \mathrm{Aut}_e(G)$. Note that while groups are mentioned in the algorithms they are implemented by their list of generators.

Now we fill in the details of how A_k is computed from A_{k-1}, see also Algorithm 3.

We construct A_k from A_{k-1} by looking at V_{k-1} and V_k and how edges in E_{k-1} connect them. We call G_{k-1} the *interior* and $(V_{k-1} \cup V_k, E_{k-1})$ the *boundary*.

Algorithm 2 A first look at the algorithm

Input: $G = (V, E)$ is a graph and $e = \{v_1, v_2\} \in E$ is an edge in that graph.
Returns: $\mathrm{Aut}_e(G)$, the group of automorphisms of G that fix e setwise.

```
1:  function AutomorphismGroup(G, e)
2:      var
3:          A:automorphism group                          ▷ A = A_{k-1}
4:          A':automorphism group                         ▷ A = A_k at the end of the loop
5:          h:integer                                     ▷ the largest i for which V_i is nonempty
6:          k:integer                                     ▷ counter from 1 to h + 1
7:      end var
8:      A := ⟨(v_1, v_2)⟩                                 ▷ A = A_0
9:      calculate h

10:     for k := 1 to h + 1 do
11:         compute A' = A_k from A = A_{k-1}
12:         A := A'                                       ▷ A = A_{k-1}(k → k + 1)
13:     end for

14:     return A                                          ▷ A = A_{h+1} = Aut_e(G)
15: end function                                          ▷ See also Algorithm 3
```

First we calculate the automorphism that pointwise stabilize V_{k-1}. These will be automorphisms of A_k. We store the Schreier Data obtained this way for later use. Then we look at the restriction of A_{k-1} to V_{k-1}.

An automorphism in A_k must transform every family of the boundary to another family of the same type. These are the automorphisms $\varphi \in A_{k-1}$ that stabilize $\mathcal{F}$, with which we mean that they stabilize the F_{k-1} parts of families (that is transforms each set F_{k-1} to a set $F_{k-1}^{\varphi} = F'_{k-1}$ of the same type). The set of families, grouped by type $\mathcal{F}$ is calculated with Algorithm 4:FamilyBlocks.

From a permutation π in this stabilizer we can get a permutation $\varphi \in A_{K-1}$ by using the Schreier Data and Algorithm 5:SchreierExtend. From this permutation and using the fact that the reduction to V_{k-1} stabilizes $\mathcal{F}$, we can extend this to a permutation in A_k with Algorithm 6:GetFamilyPerm.

We now construct A_k with the generators of the interior automorphism group and the extended generators of the stabilizer. Since all permutations of the vertices of V_k in the same family are automorphisms we also add generators for these.

3.4.1 Constructing the families

In the function FamilyBlocks, described in Algorithm 4, the families are calculated and grouped according to their types. A family F of type other than $(2, 0)$ has a nonempty $F_k \subseteq V_k$. Furthermore each vertex in V_k can only be in one family. Therefore the families of type other than $(2, 0)$ can be found by looking at all the vertices in V_k. Each vertex is marked once the corresponding family is known. A family of type $(2, 0)$ corresponds to an edge in E_{k-1} between two vertices of V_{k-1}. These families can be found by checking all edges in E_{k-1}.

3.4.2 From $\pi \in A_{k-1}|_{V_{k-1}}$ to $\varphi \in A_{k-1}$

With the construction of the pointwise stabilizer of V_{k-1} by A_{k-1} , H, Schreier data is generated. This data can be used to extend $\pi \in A_{k-1}|_{V_{k-1}}$ to A_{k-1}. In this section we describe Algorithm 5.

This is done by constructing a permutation $\phi = \pi * \pi'$, where $\pi' \in A_{k-1}$, such that $\phi(v) = v$, for $v \in V_{k-1}$. The extension of π can then be formed by calculating $\phi^{-1} * \pi = (\pi * \pi')^{-1} * \pi = \pi'^{-1} * \pi^{-1}\pi = \pi' \in \mathrm{Aut}_e(G_{k-1})$.

Algorithm 3 Automorphism of connected graphs leaving an edge invariant

Input: $G = (V, E)$ is a graph and $e = \{v_1, v_2\} \in E$ is an edge in that graph.
Returns: $\mathrm{Aut}_e(G)$, the group of automorphisms of G that fix e setwise.

function AUTOMORPHISMGROUP(G, e)
  var
    A:automorphism group ▷ $A = A_{k-1}$
    A':automorphism group ▷ $A = A_k$ at the end of the loop
    h:integer ▷ the largest i, for which V_i is nonempty
    $V_0, \ldots, V_h$:sets of vertices ▷ the value of these sets is as defined in 3.2
    $E_0, \ldots, E_{h+1}$:sets of edges ▷ the value of these sets is as defined in 3.2
    k:integer ▷ counter from 1 to $h + 1$
    $\mathcal{F}$:set of families ▷ this set contains the set of families between V_{k-1} and V_k as described in 3.2
    F:family ▷ $F \in \mathcal{F}$
    H:permutation group ▷ subgroup of $A_{k-1}\,|_{V_{k-1}}$
    π:permutation ▷ $\pi \in H$
    φ:permutation ▷ $\varphi \in Aut_e(G_{k-1})$
    χ:permutation ▷ permutation on V_k
    sd:Schreier data ▷ Schreier data from calculating the pointwise stabilizer of V_{k-1} by A_{k-1}.
  end var
  $A := \langle (v_1, v_2) \rangle$ ▷ $A = A_0$
  calculate h, $V_0, \ldots, V_h$, and $E_0, \ldots, E_h$

  for $k := 1$ **to** $h + 1$ **do**
    $\mathcal{F} :=$ FAMILYBLOCKS(V_{k-1}, V_k, E_{k-1}) ▷ (see FAMILYBLOCKS)
    let H be the stabilizer in $A|_{V_{k-1}}$ that stabilizes $\mathcal{F}$ pointwise by type ▷ stabilized one-by-one per type
    initialize A' with the pointwise stabilizer of V_{k-1} by A and store the calculated Schreier data sd

    for each generator π of H **do**
      $\varphi :=$ SCHREIEREXTEND(π, sd) ▷ $\varphi \in \mathrm{Aut}_e(G_{k-1})$ s.t. $\varphi|_{V_k} = \pi$, see Algorithm 5
      if $k \leq h$ **then**
        $\chi :=$ GETFAMILYPERM($F, V_k, V_{k-1}, E_{k-1}, \varphi$) ▷ Algorithm 6
        $A' := \langle A', \chi \rangle$
      else ▷ $V_k = V_{h+1}$ is empty
        $A' := \langle A', \varphi \rangle$
      end if
    end for

    for each family F **do** ▷ $F = (F_{k-1}, F_k)$
      if $|F_k| \geq 2$ **then**
        $A' := \langle A', (u_1, u_2, \ldots, u_t), (u_1, u_2) \rangle$ ▷ $F_k = \{u_1, \ldots, u_t\}$
      end if
    end for ▷ A' is now $\mathrm{Aut}_e(G_k)$
    $A := A'$ ▷ $A = \mathrm{Aut}_e(G_{k-1}(k := k + 1))$
  end for ▷ $A = \mathrm{Aut}_e(G_{h+1})$

  return A
end function

Algorithm 4 Constructing the families

Input: This function is called from AUTOMORPHISMGROUP, where G is a graph and e an edge in that graph. Conform the definitions in 3.2, V_{k-1} is the set of vertices at distance $k-1$ to e, V_k is the set of vertices at distance k of e and E_{k-1} is the set of edges between V_{k-1} and $V_{k-1} \cup V_k$.

Returns: $\mathcal{F}$, the set of families of the edges, grouped by type.

1: **function** FAMILYBLOCKS(V_{k-1}, V_k, E_{k-1})
2: **var**
3: $\mathcal{F}$:list of sets of families, ordered by type
4: a:vertex
5: v:vertex
6: w:vertex
7: $F = (F_{k-1}, F_k)$:family ▷ family being constructed, $w \in F_{k-1}$
8: a boolean list of the vertices in V_k indicating whether they are *marked*
9: **end var**
10: $\mathcal{F} := []$ ▷ $\mathcal{F}$ is the set of families to form
▷ find all families of type $\neq (0, 2)$
11: **for** $w \in V_k$ not yet marked **do**
12: $F_{k-1} :=$ ANCESTRY(w)
13: choose $a \in F_{k-1}$ ▷ possible, because ANCESTRY(w) $\neq []$
14: $F_k := \{w\}$

15: **for** $\{a, v\} \in E_{k-1}$ **do**
16: **if** $(v \in V_k)$ **and** $(v \neq w)$ **and** ANCESTRY(v) $= F_{k-1}$ **then**
17: add v to F_k
18: **end if**
19: **end for**

20: add (F_{k-1}, F_k) to the set of families of type (i, j) in $\mathcal{F}$ ▷ where $(i = |F_{k-1}|, j = |F_k|)$ is the family type
21: mark the vertices in F_k ▷ each vertex in F_k is in only one family
22: **end for**

23: **for** all edges $\{v, w\} \in E_{k-1}$ **do**
24: **if** $v \in V_{k-1}$ **and** $w \in V_{k-1}$ **then**
25: add $(\{v, w\}, \emptyset)$ to F
26: **end if**
27: **end for**

28: **return** $\mathcal{F}$ ▷ list of families in the form $(U_{k-1} \subseteq V_{k-1}, U_k \subseteq V_k)$, grouped by type.
29: **end function**

To construct $\phi = \pi * \pi'$, we initialize with $\phi = \pi$. Suppose the vertices with index $j < i$ in V_{k-1} are already stabilized then we can stabilize vertex i by looking at the vertex $v = V_{k-1}[i]$ and its image $v' = v^\phi$. From the Schreier data we find a $\pi' \in \mathrm{Aut}_e(G_{k-1})$ such that $v'^{\pi'} = v$. Note that we can only find such a π' if an extension of π to $\mathrm{Aut}_e(G_{k-1})$ exists. By doing this for all vertices we find the ϕ that stabilizes all $v \in V_{k-1}$.

Algorithm 5 From $\pi \in \mathrm{Aut}_e(G_{k-1})|_{V_{k-1}}$ to $\varphi \in \mathrm{Aut}_e(G_{k-1)}$

Input: $\pi \in \mathrm{Aut}_e(G_{k-1})|_{V_{k-1}}$, V_{k-1} a list of vertices, in the order corresponding to the Schreier Data sd of V_{k-1} in G_{k-1}.
Returns: $\varphi \in \mathrm{Aut}_e(G_{k-1})$, such that $\varphi|_{V_{k-1}} = \pi$.

1: **function** SchreierExtend(π, sd, V_{k-1})
2: **var**
3: φ:permutation
4: φ':permutation
5: i:integer ▷ index of V_{k-1}
6: v:vertex ▷ $v = V_{k-1}[i]$
7: v':vertex ▷ $v' = v^\varphi$
8: **end var**
9: $\varphi = \pi$ ▷ invariant $\varphi = \pi' * \pi$, with $\pi' \in \mathrm{Aut}_e(G_{k-1})$
▷ invariant $V_{k-1}[j]^\varphi = V_{k-1}[j]$ for $0 < j < i$
10: **for** $i \in 1, \ldots, |V_{k-1}|$ **do**
11: $v = V_{k-1}[i]$
12: $v' := v^\varphi$
13: From sd corresponding to v get $\varphi' \in \mathrm{Aut}_e(G_{k-1})$ such that $v'^{\varphi'} = v$.
14: $\varphi := \varphi * \varphi'$ ▷ $v^{\varphi*\varphi'} = (v^\varphi)^{\varphi'} = v'^{\varphi'} = v$
▷ $\varphi = \pi * \pi'$ holds for $\pi' := \pi'\varphi'$
15: **end for** ▷ $\varphi = \pi' * \pi$ and $V_{k-1}[j]^\varphi = V_{k-1}[j]$ for $0 < j \leq |V_{k-1}|$
16: **return** $\varphi^{-1} * \pi$ ▷ $(\pi * \pi')^{-1} * \pi = \pi'^{-1} * \pi^{-1} * \pi = \pi'^{-1} \in \mathrm{Aut}_e(G_{k-1})$
▷ furthermore $v^{(\pi'*\pi)^{-1}*\pi} = (v^{(\pi'*\pi)^{-1}})^\pi = v^\pi$.
17: **end function**

3.4.3 Extending an automorphism to the boundary

To extend φ we need to find an image for the vertices in V_k grouped in F_k such that edges in E_k are transformed into edges. If the families (F_{k-1}, F_k) are transferred to families (F'_{k-1}, F'_k) then it is clear to see that edges in that family are indeed transformed to edges. What this procedure does is for each F_{k-1} calculate the F'_{k-1}, find the corresponding F'_k and set the image of the vertices in F_k to those in F'_k (since we want a permutation we use a bijective map here).

3.5 Example

We demonstrate how the proof that two certain graphs are not isomorphic is produced by this algorithm with an extended example. The proof generated can be found at the end of the chapter. The algorithm has been implemented in GAP [9]. Given a graph it gives the automorphism group that leaves an edge invariant and a proof. It is also available from the proof assistant to use this algorithm to check graph isomorphism.

The stabilizer proofs are sometime long, because proving the pointwise stabilizer of a set involves proving several stabilizers for one point. It is possible to suppress stabilizer proofs though, so they are invisible unless a user requests them.

Luks' algorithm gives us the isomorphism group of a graph leaving an edge invariant. We can 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 and running Luks' algorithm on that graph and that edge.

Algorithm 6 Extend $\varphi \in \mathrm{Aut}_e(G_{k-1})$ to a $\varphi * \psi \in \mathrm{Aut}_e(G_k)$

Input: This function is called from AutomorphismGroup where G is a graph and e is an edge of G. Conform the definitions in 3.2, $\mathcal{F}$ is the set of families (grouped by type), V_k is the set of vertices at distance k of e, V_{k-1} is the set of vertices at distance $k-1$ of e, E_{k-1} is the set of edges between V_{k-1} and $V_{k-1} \cup V_k$. Furthermore $\varphi \in \mathrm{Aut}_e(G_{k-1})$ stabilizes $\mathcal{F}$, i.e. transforms a family to a family of the same type.

Returns: $\varphi * \psi \in Aut_e(G_k)$, such that $\psi|_{G_{k-1}} = ()$.

1: **function** GetFamilyPerm($\mathcal{F}, V_k, V_{k-1}, E_{k-1}, \varphi$)
2: **var**
3: ψ:permutation
4: F:family
5: F'_{k-1}:set of vertices ▷ $F'_{k-1} = \varphi F_{k-1} \subseteq V_{k-1}$.
6: F'_k:set of vertices ▷ $F'_k \subseteq V_k$ and $F' = (F'_{k-1}, F'_k)$ is a family
7: **end var**
▷ now we construct the permutation ψ on V_k. For each $v \in V_k$ we need to find a valid image. This can be done by looping through the families (each vertex in V_k is in exactly one F_k).
8: **for each** family $F \in \mathcal{F}$ **do**
9: $F'_{k-1} := \varphi F_{k-1}$ (on sets)
10: calculate a corresponding F'_k ▷ possible since φ stabilizes $\mathcal{F}$
11: add to ψ as images of F_k the vertices F'_k
12: **end for** ▷
13: **return** $\varphi * \psi$ ▷ where φ stabilized the V_{k-1} parts of $\mathcal{F}$, $\varphi * phi$ now stabilizes the whole $\mathcal{F}$
14: **end function**

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. To calculate the orbits the automorphism group can be calculated beforehand. Note that here it is only necessary to prove that all generators are automorphisms and not that we have all of them because a split orbit only leads to an extra invocation.

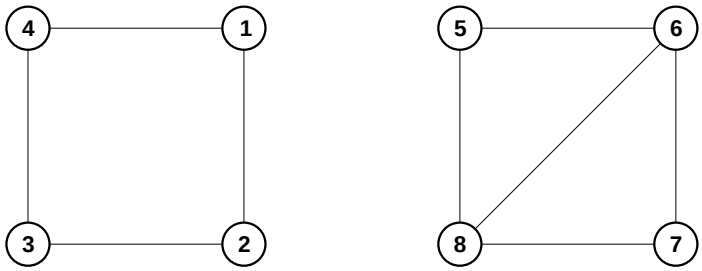

Figure 3.7: Two graphs

We now give a worked example with the two graphs in Figure 3.7. We have already renumbered the vertices of the second graph so that they are disjoint. Note that these can be easily told apart by the number of edges, but we use these small graphs for simplicity. 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 3.8.

We initialize $A_0 = \langle (1,5) \rangle$. Now we want to calculate A_1. Remember that extending from A_{k-1} to A_k is done in three steps. First we calculate the automorphisms in the interior, then we extend the algorithms acting on the boundary and finally we add the automorphisms on V_k that leave V_{k-1} invariant.

Since all vertices are on the boundary, there are no non-trivial automorphisms on the interior.

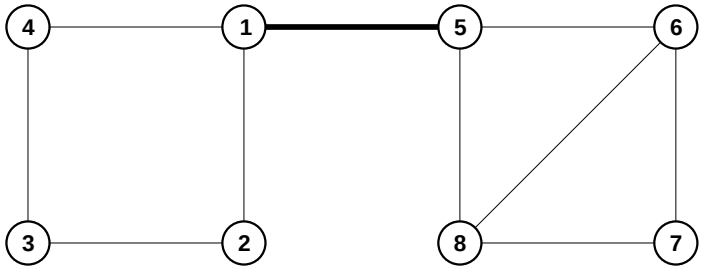

Figure 3.8: Luks' algorithm with the edge bold

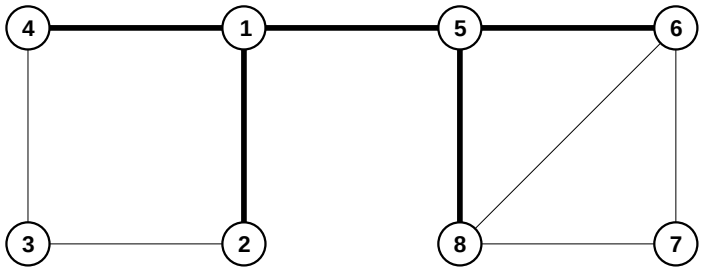

Figure 3.9: Luks' algorithm with E_0 bold

Now $(1,5)$ has to be extended. We look at the edges E_0 connecting V_0 to V_1, see Figure 3.9, and split them into families. There are three families: two families of type $(1,2)$: $(F_0 = \{1\}, F_1 = \{2,4\})$ and $(F_0 = \{5\}, F_1 = \{6,8\})$ and one of type $(0,2)$: $(F_0 = \{1,5\}, F_1 = \emptyset)$. This leads to the following structure to stabilize such that the families are preserved: $(\{\{1\},\{5\}\},\{\{1,5\}\})$. A proof that the stabilizer is $\langle(1,5)\rangle$, is given on lines 107–114 of the long proof, following the methods of Section 3.3.

Since there is no interior, $(1,5) \in A_0$. We now extend it to A_1 as follows: the family $(F_0 = \{1\}, F_1 = \{2,4\})$ is mapped to the family $(F_0 = \{5\}, F_1\{6,8\})$ and that family is mapped again to the first family. The permutation that transforms the elements in V_1 in such a way that the families are preserved is $(2,6)(4,8)$. The family $(F_0 = \{1,5\}, F_1 = \emptyset)$ is mapped to itself. So no extension is needed for this family. We then multiply $(1,5)$ by $(2,6)(4,8)$ to get $(1,5)(2,6)(4,8) \in A_1$.

In the last step we add the automorphisms inside the families. They are $(2,4)$ from the family $(F_0 = \{1\}, F_1 = \{2,4\})$ and $(6,8)$ from the family $(F_0 = \{5\}, F_1 = \{6,8\})$. A_1 now becomes $\langle(1,5)(2,6)(4,8),(2,4),(6,8)\rangle$.

Next we calculate A_2 from A_1 by looking at the vertices $V_1 = \{2,4,6,8\}$ at distance 1 and 2, see also Figure 3.10. First we look for automorphisms on the interior that stabilize V_1 vertexwise. There are no such automorphisms in A_1 as proved in lines 134–239 of the long version. Then we restrict A_1 to V_1. We find that $A_1 \mid_{V_1} = \langle(2,6)(4,8),(2,4),(6,8)\rangle$.

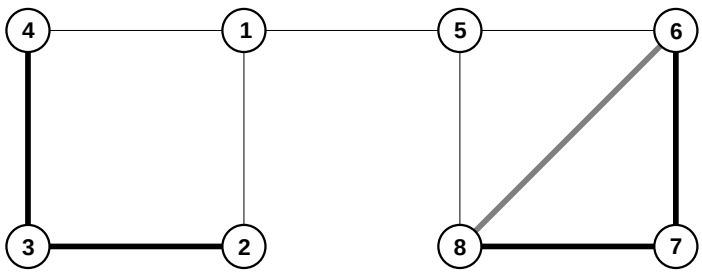

Figure 3.10: Luks' algorithm with E_1 bold

Now we look at the families. There are three families. Two of type $(2,1)$: $(F_1 = \{2,4\}, F_2 =$

$\{3\}$) and ($F_1 = \{6,8\}, F_2 = \{7\}$) and one of type $(0,2)$: ($F_1 = \{6,8\}, F_2 = \emptyset$). This leads to the following structure to stabilize, so that the families are preserved: $(\{\{2,4\},\{6,8\}\},\{\{6,8\}\})$. The stabilizer is $\langle(2,4),(6,8)\rangle$. By using Schreier trees we find the corresponding group in A_1: $\langle(2,4),(6,8)\rangle$. Both permutations preserve all families and do not need to be extended. In the final step we notice that the F_2 parts of the families are either empty or only have one member: this means that there are no automorphisms to add in this step.

Clearly A_2 does not contain any permutation that maps 1 to 5 (and 5 to 1). Since such a permutation cannot be added later in the algorithm it is clear that there exists no automorphism that maps the graphs to each other. We can now conclude that the graphs are nonisomorphic.

3.6 Modifications for colors

Luks' algorithm is for uncolored graphs, but it is possible to extend it to colored graphs. One possibility would be to add colors in the type of a family. Unfortunately this would mean that the possible number of types of families would be $|C|$ times larger. It is also possible to only allow vertices with the ancestry *and* the same color in the same family. This would lead to $|C|$ times as much families, where $|C|$ is the number of colors used.

Algorithm output

Here is the produced proof from the example in Section 3.5.

Long version

This is the full version of the proof, including orbit and stabilizer proofs.

```
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:
    Proposition: the orbit of x = 1 under A, the group generated by a1 = (2,4) and a2 = (1,3), is X = [ 1, 3 ].
    Proof:
      By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A leaves X
      invariant. It remains to show that the points in X are images of x under elements of A:
        1 = 1^()
        3 = 1^(1,3) = 1^a2
    QED(orbit of x = 1)
    Proposition: the orbit of x = 2 under A, the group generated by a1 = (2,4) and a2 = (1,3), is X = [ 2, 4 ].
    Proof:
      By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A leaves X
      invariant. It remains to show that the points in X are images of x under elements of A:
        2 = 2^()
        4 = 2^(2,4) = 2^a1
    QED(orbit of x = 2)
    It is straightforward to verify that the orbits [ 1, 3 ] and [ 2, 4 ] are disjoint and their union is [ 1, 2, 3, 4 ], so we are done.
  QED(all orbits)
  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.
        First we split V and E depending on the distance to e. The proof as done is by checking.
        Proposition: on the graph with vertices V=[1..8] and edges E=[ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8
        ], [ 7, 8 ] ], the edge e=[ 1, 5 ] splits the vertices and edges thus:
          V is split in V_i, sets of vertices at distance i to e:
          V_0 = [ 1, 5 ],
          V_1 = [ 2, 4, 6, 8 ] and
          V_2 = [ 3, 7 ].
          E is split in E_i, sets of edges between vertices at distance i to e and vertices of distance i or i+1 to e:
          E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 5, 6 ], [ 5, 8 ] ] and
          E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8 ] ].
        Proof:
          It suffices to verify that (i) the V_i and E_i are disjoint, (ii) the union of the V_i is V, (iii) the union of the E_i is E, (iv) the edges in E_i
          are between a vertex in V_i and one in V_i or V_i+1 and (v) V_i is at distance at most i to e.
        QED(VertexSets)
        A_0 is [ (1,5) ].
        Proposition: A_1 = [ (6,8), (2,4), (1,5)(2,6)(4,8) ].
        Proof:
          We now calculate A_1 in three steps.
          Step 1: Proposition: the vertexwise stabilizer (H) of A_0(=A) on V_0 is the group generated by ().
            Proof:
              Proposition: Let A be the group generated by a1 = (1,5). Let A_([ 1 ]) be the trivial group <()>. Then the stabilizer of x = 1 in A is A_([ 1 ]).
              Proof:
```

```
        First we check that A_([ 1 ]) is a subgroup of A:
        Proposition: Let A_([ 1 ]) be the trivial group <()>. Let A be the group generated by a1 = (1,5). Then H is a subgroup of G.
        Proof:
          It suffices to verify that each of the generators of A_([ 1 ]) belongs to A. Below we give a word in the generators of A for each generator
          of A_([ 1 ]).
          Proof:
          QED(proof by generator)
          This establishes that each generator of A_([ 1 ]) belongs to A, and so A_([ 1 ]) is a subgroup of A.
        QED(issubgroup)
        Next we determine the Schreier data for A at x:
        Proposition: the Schreier data for A, the group generated by a1 = (1,5), at x = 1 is the table
          [ 1, 5 ]
          [ 0, 1 ]
          [ 0, 1 ]
          It has three rows, the first of which represents the A-orbit X of 1.
          Proposition: the orbit of x = 1 under A, the group generated by a1 = (1,5), is X = [ 1, 5 ].
          Proof:
            By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A
            leaves X invariant. It remains to show that the points in X are images of x under elements of A:
              1 = 1^()
              5 = 1^(1,5) = 1^a1
          QED(orbit of x = 1)
          To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
          verify that the image of x under the (-v)-th generator of A equals z if v<0, and that the image of x under the inverse of the v-the
          generator equals z if v>0.
           1 () = 0
           5 (1,5) = 1
          To finish, note that each z occurs earlier than x in the orbit list.
        QED(schreierdata)
        From the Schreier data we find that the Schreier elements are as follows:
        t(1) =  ()  = ()
        t(5) =  a1^-1  = (1,5)
        Finally we check that each Schreier generator is in A_([ 1 ]):
        Schreier gen(1,1)  = a1 t(a1)^(-1)  = () = ()
        Schreier gen(5,1)  = () t(())^(-1)  = a1^2 = ()
        By Schreier's lemma, A_([ 1 ]) is the stabilizer in A of x.
      QED(stabilizer)
      It is easy to see that the point 5 is not moved by any permutation from A_([ 1 ]). Therefore H = A_([ 1 ]), the group generated by ().
    QED(pointwise stabilizer)
  QED(Step 1)
  Step 2: Proposition: the extension of the automorphism that doesn't leave V_0 invariant is (1,5)(2,6)(4,8)
    The reduction of A_0 to V_0 is [ (1,5) ]
    It is easy to verify that these are the cycles of permutations in A_0 that contain only vertices in V_0 (note that a cycle containing one vertex
    of V_0 must contain only vertices in V_0).
    Now we calculate the families.
      The families between V_0 = [ 1, 5 ] and V_1 = [ 2, 4, 6, 8 ], that are connected by edges E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 5, 6 ], [ 5, 8
      ] ] are:
      The families of type (2, 0) are [ [ [ 1, 5 ], [ ] ] ].
      The families of type (1, 2) are [ [ [ 1 ], [ 2, 4 ] ], [ [ 5 ], [ 6, 8 ] ] ].
       For this to be the set of families it is necessary that each vertex of V_1 occurs exactly once, the vertices in a family from V_1 are connected
      to those in that family from V_0 and these connections form all the edges in E_0. This can all be easily checked.
    QED(FamilyBlocks)
    Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
    of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
    Proof:
      It is easy to see that the point [ [ 1, 5 ] ] is not moved by any permutation from G. Therefore G_([ [ [ 1, 5 ] ] ]) = G, the group generated by
      (1,5).
      It is easy to see that the point [ [ 1 ], [ 5 ] ] is not moved by any permutation from G_([ [ [ 1, 5 ] ] ]). Therefore H = G_([ [ [ 1, 5 ] ] ]),
      the group generated by (1,5).
    QED(pointwise stabilizer)
    Now extend each generator of the stabilizer to A_0.
    The extension of (1,5) is (1,5). Now we show that (1,5) is in A_0:
      The element a = (1,5) can be written as follows as a word in the generators of A_0: a = a_01, and so belongs to A_0.
    QED(isin)
    The extension of (1,5) to a permutation that leaves E_0 invariant is (1,5)(2,6)(4,8)
    Proof:
      The permutation (1,5)(2,6)(4,8)|_G_0 = (1,5) and (1,5)(2,6)(4,8) is an automorphism of G_0. This can all be easily verified.
    QED(GetFamilyPerm)
  QED(Step 2)
    These are formed by the symmetric groups of the vertices in V_1 for each family.
  Step 3: Proposition: the automorphisms on V_1 that leave V_0 invariant are (2,4) and (6,8).
    The family [ [ 1 ], [ 2, 4 ] ] adds the generators [ (2,4) ].
    The family [ [ 5 ], [ 6, 8 ] ] adds the generators [ (6,8) ].
  QED(Step 3)
  By combining these steps we find that A_1 is [ (6,8), (2,4), (1,5)(2,6)(4,8) ]
QED(A_0->A_1)
Proposition: A_2 = [ (6,8), (2,4) ].
Proof:
  We now calculate A_2 in three steps.
  Step 1: Proposition: the vertexwise stabilizer (H) of A_1(=A) on V_1 is the group generated by ().
    Proof:
      Proposition: Let A be the group generated by a1 = (6,8), a2 = (2,4) and a3 = (1,5)(2,6)(4,8). Let A_([ 2 ]) be the group generated by a_([ 2 ])1
      = (6,8). Then the stabilizer of x = 2 in A is A_([ 2 ]).
      Proof:
        First we check that A_([ 2 ]) is a subgroup of A:
        Proposition: Let A_([ 2 ]) be the group generated by a_([ 2 ])1 = (6,8). Let A be the group generated by a1 = (6,8), a2 = (2,4) and a3 =
        (1,5)(2,6)(4,8). Then H is a subgroup of G.
        Proof:
          It suffices to verify that each of the generators of A_([ 2 ]) belongs to A. Below we give a word in the generators of A for each generator
          of A_([ 2 ]).
          Proof:
            a_([ 2 ])1 = (6,8) = a1.
          QED(proof by generator)
          This establishes that each generator of A_([ 2 ]) belongs to A, and so A_([ 2 ]) is a subgroup of A.
        QED(issubgroup)
        Next we determine the Schreier data for A at x:
        Proposition: the Schreier data for A, the group generated by a1 = (6,8), a2 = (2,4) and a3 = (1,5)(2,6)(4,8), at x = 2 is the table
          [ 2, 4, 6, 8 ]
          [ 0, 2, 3, 3 ]
          [ 0, 2, 2, 4 ]
          It has three rows, the first of which represents the A-orbit X of 2.
          Proposition: the orbit of x = 2 under A, the group generated by a1 = (6,8), a2 = (2,4) and a3 = (1,5)(2,6)(4,8), is X = [ 2, 4, 6, 8 ].
          Proof:
            By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A
            leaves X invariant. It remains to show that the points in X are images of x under elements of A:
              2 = 2^()
```

```
                    4 = 2^(2,4) = 2^a2
                    6 = 2^(1,5)(2,6)(4,8) = 2^a3
                    8 = 2^(1,5)(2,8,4,6) = 2^a2*a3
                  QED(orbit of x = 2)
                  To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
                  verify that the image of x under the (-v)-th generator of A equals z if v<0, and that the image of x under the inverse of the v-the
                  generator equals z if v>0.
                   2 () = 0
                   4 (2,4) = 2
                   6 (1,5)(2,6)(4,8) = 2
                   8 (1,5)(2,6)(4,8) = 4
                  To finish, note that each z occurs earlier than x in the orbit list.
                QED(schreierdata)
                From the Schreier data we find that the Schreier elements are as follows:
                t(2) =  ()  = ()
                t(4) =  a2^-1  = (2,4)
                t(6) =  a3^-1  = (1,5)(2,6)(4,8)
                t(8) =  a3^-1*a2^-1  = (1,5)(2,6,4,8)
                Finally we check that each Schreier generator is in A_([ 2 ]):
                Schreier gen(2,1)  = a1 t(a1)^(-1)  = a1 = a_([ 2 ])1
                Schreier gen(2,2)  = a2 t(a2)^(-1)  = () = ()
                Schreier gen(2,3)  = a3 t(a3)^(-1)  = () = ()
                Schreier gen(4,1)  = a2^-1*a1 t(a2^-1*a1)^(-1)  = a2*a1*a2^-1 = a_([ 2 ])1
                Schreier gen(4,2)  = () t(())^(-1)  = a2^2 = ()
                Schreier gen(4,3)  = a2^-1*a3 t(a2^-1*a3)^(-1)  = () = ()
                Schreier gen(6,1)  = a3^-1*a1 t(a3^-1*a1)^(-1)  = a3*a1*a3^-1*a2^-1 = ()
                Schreier gen(6,2)  = a3^-1*a2 t(a3^-1*a2)^(-1)  = a3*a2*a3^-1 = a_([ 2 ])1
                Schreier gen(6,3)  = () t(())^(-1)  = a3^2 = ()
                Schreier gen(8,1)  = a3^-1*a2^-1*a1 t(a3^-1*a2^-1*a1)^(-1)  = a2*a3*a1*a3^-1 = ()
                Schreier gen(8,2)  = a3^-1 t(a3^-1)^(-1)  = a2*a3*a2*a3^-1*a2^-1 = a_([ 2 ])1
                Schreier gen(8,3)  = a3^-1*a2^-1*a3 t(a3^-1*a2^-1*a3)^(-1)  = a2*a3^2*a2^-1 = ()
                By Schreier's lemma, A_([ 2 ]) is the stabilizer in A of x.
              QED(stabilizer)
              It is easy to see that the point 4 is not moved by any permutation from A_([ 2 ]). Therefore A_([ 2, 4 ]) = A_([ 2 ]), the group generated by
              (6,8).
              Proposition: Let A_([ 2, 4 ]) be the group generated by a_([ 2, 4 ])1 = (6,8). Let A_([ 2, 4, 6 ]) be the trivial group <()>. Then the
              stabilizer of x = 6 in A_([ 2, 4 ]) is A_([ 2, 4, 6 ]).
              Proof:
                First we check that A_([ 2, 4, 6 ]) is a subgroup of A_([ 2, 4 ]):
                Proposition: Let A_([ 2, 4, 6 ]) be the trivial group <()>. Let A_([ 2, 4 ]) be the group generated by a_([ 2, 4 ])1 = (6,8). Then H is a
                subgroup of G.
                Proof:
                  It suffices to verify that each of the generators of A_([ 2, 4, 6 ]) belongs to A_([ 2, 4 ]). Below we give a word in the generators of A_([
                  2, 4 ]) for each generator of A_([ 2, 4, 6 ]).
                  Proof:
                  QED(proof by generator)
                  This establishes that each generator of A_([ 2, 4, 6 ]) belongs to A_([ 2, 4 ]), and so A_([ 2, 4, 6 ]) is a subgroup of A_([ 2, 4 ]).
                QED(issubgroup)
                Next we determine the Schreier data for A_([ 2, 4 ]) at x:
                Proposition: the Schreier data for A_([ 2, 4 ]), the group generated by a_([ 2, 4 ])1 = (6,8), at x = 6 is the table
                  [ 6, 8 ]
                  [ 0, 1 ]
                  [ 0, 6 ]
                  It has three rows, the first of which represents the A_([ 2, 4 ])-orbit X of 6.
                  Proposition: the orbit of x = 6 under A_([ 2, 4 ]), the group generated by a_([ 2, 4 ])1 = (6,8), is X = [ 6, 8 ].
                  Proof:
                    By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A_([
                    2, 4 ]) leaves X invariant. It remains to show that the points in X are images of x under elements of A_([ 2, 4 ]):
                      6 = 6^()
                      8 = 6^(6,8) = 6^a_([ 2, 4 ])1
                  QED(orbit of x = 6)
                  To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
                  verify that the image of x under the (-v)-th generator of A_([ 2, 4 ]) equals z if v<0, and that the image of x under the inverse of the
                  v-the generator equals z if v>0.
                   6 () = 0
                   8 (6,8) = 6
                  To finish, note that each z occurs earlier than x in the orbit list.
                QED(schreierdata)
                From the Schreier data we find that the Schreier elements are as follows:
                t(6) =  ()  = ()
                t(8) =  a_([ 2, 4 ])1^-1  = (6,8)
                Finally we check that each Schreier generator is in A_([ 2, 4, 6 ]):
                Schreier gen(6,1)  = a_([ 2, 4 ])1 t(a_([ 2, 4 ])1)^(-1)  = () = ()
                Schreier gen(8,1)  = () t(())^(-1)  = a_([ 2, 4 ])1^2 = ()
                By Schreier's lemma, A_([ 2, 4, 6 ]) is the stabilizer in A_([ 2, 4 ]) of x.
              QED(stabilizer)
              It is easy to see that the point 8 is not moved by any permutation from A_([ 2, 4, 6 ]). Therefore H = A_([ 2, 4, 6 ]), the group generated by
              ().
            QED(pointwise stabilizer)
          QED(Step 1)
          Step 2: Proposition: the extensions of the automorphisms that don't leave V_1 invariant are (6,8) and (2,4)
            The reduction of A_1 to V_1 is [ (6,8), (2,4), (2,6)(4,8) ]
            It is easy to verify that these are the cycles of permutations in A_1 that contain only vertices in V_1 (note that a cycle containing one vertex
            of V_1 must contain only vertices in V_1).
            Now we calculate the families.
              The families between V_1 = [ 2, 4, 6, 8 ] and V_2 = [ 3, 7 ], that are connected by edges E_1 = [ [ [ 2, 3 ], [ 3, 4 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8
              ] ] are:
              The families of type (2, 0) are [ [ [ 6, 8 ], [ ] ] ].
              The families of type (2, 1) are [ [ [ 2, 4 ], [ 3 ] ], [ [ 6, 8 ], [ 7 ] ] ].
               For this to be the set of families it is necessary that each vertex of V_2 occurs exactly once, the vertices in a family from V_2 are connected
              to those in that family from V_1 and these connections form all the edges in E_1. This can all be easily checked.
            QED(FamilyBlocks)
            Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
            of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
            Proof:
              Proposition: Let G be the group generated by g1 = (6,8), g2 = (2,4) and g3 = (2,6)(4,8). Let G_([ [ [ 6, 8 ] ] ]) be the group generated by g_([
              [ [ 6, 8 ] ] ])1 = (2,4) and g_([ [ [ 6, 8 ] ] ])2 = (6,8). Then the stabilizer of x = [ [ 6, 8 ] ] in G is G_([ [ [ 6, 8 ] ] ]).
              Proof:
                First we check that G_([ [ [ 6, 8 ] ] ]) is a subgroup of G:
                Proposition: Let G_([ [ [ 6, 8 ] ] ]) be the group generated by g_([ [ [ 6, 8 ] ] ])1 = (2,4) and g_([ [ [ 6, 8 ] ] ])2 = (6,8). Let G be the
                group generated by g1 = (6,8), g2 = (2,4) and g3 = (2,6)(4,8). Then H is a subgroup of G.
                Proof:
                  It suffices to verify that each of the generators of G_([ [ [ 6, 8 ] ] ]) belongs to G. Below we give a word in the generators of G for each
                  generator of G_([ [ [ 6, 8 ] ] ]).
                  Proof:
                    g_([ [ [ 6, 8 ] ] ])1 = (2,4) = g2;
```

```
                    g_([ [ [ 6, 8 ] ] ])2 = (6,8) = g1.
                  QED(proof by generator)
                  This establishes that each generator of G_([ [ [ 6, 8 ] ] ]) belongs to G, and so G_([ [ [ 6, 8 ] ] ]) is a subgroup of G.
                QED(issubgroup)
                Next we determine the Schreier data for G at x:
                Proposition: the Schreier data for G, the group generated by g1 = (6,8), g2 = (2,4) and g3 = (2,6)(4,8), at x = [ [ 6, 8 ] ] is the table
                  [ [ [ 6, 8 ] ], [ [ 2, 4 ] ] ]
                  [ 0, 3 ]
                  [ [ [ 0, 0 ] ], [ [ 6, 8 ] ] ]
                  It has three rows, the first of which represents the G-orbit X of [ [ 6, 8 ] ].
                  Proposition: the orbit of x = [ [ 6, 8 ] ] under G, the group generated by g1 = (6,8), g2 = (2,4) and g3 = (2,6)(4,8), is X = [ [ [ 2, 4 ]
                  ], [ [ 6, 8 ] ] ].
                  Proof:
                    By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of G
                    leaves X invariant. It remains to show that the points in X are images of x under elements of G:
                      [ [ 2, 4 ] ] = [ [ 6, 8 ] ]^(2,6)(4,8) = [ [ 6, 8 ] ]^g3
                      [ [ 6, 8 ] ] = [ [ 6, 8 ] ]^()
                  QED(orbit of x = [ [ 6, 8 ] ])
                  To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
                  verify that the image of x under the (-v)-th generator of G equals z if v<0, and that the image of x under the inverse of the v-the
                  generator equals z if v>0.
                   [ [ 6, 8 ] ] () = [ [ 0, 0 ] ]
                   [ [ 2, 4 ] ] (2,6)(4,8) = [ [ 6, 8 ] ]
                  To finish, note that each z occurs earlier than x in the orbit list.
                QED(schreierdata)
                From the Schreier data we find that the Schreier elements are as follows:
                t([ [ 6, 8 ] ]) =  ()  = ()
                t([ [ 2, 4 ] ]) =  g3^-1  = (2,6)(4,8)
                Finally we check that each Schreier generator is in G_([ [ [ 6, 8 ] ] ]):
                Schreier gen([ [ 6, 8 ] ],1)  = g1 t(g1)^(-1)  = g1 = g_([ [ [ 6, 8 ] ] ])2
                Schreier gen([ [ 6, 8 ] ],2)  = g2 t(g2)^(-1)  = g2 = g_([ [ [ 6, 8 ] ] ])1
                Schreier gen([ [ 6, 8 ] ],3)  = g3 t(g3)^(-1)  = () = ()
                Schreier gen([ [ 2, 4 ] ],1)  = g3^-1*g1 t(g3^-1*g1)^(-1)  = g3*g1*g3^-1 = g_([ [ [ 6, 8 ] ] ])1
                Schreier gen([ [ 2, 4 ] ],2)  = g3^-1*g2 t(g3^-1*g2)^(-1)  = g3*g2*g3^-1 = g_([ [ [ 6, 8 ] ] ])2
                Schreier gen([ [ 2, 4 ] ],3)  = () t(())^(-1)  = g3^2 = ()
                By Schreier's lemma, G_([ [ [ 6, 8 ] ] ]) is the stabilizer in G of x.
              QED(stabilizer)
              It is easy to see that the point [ [ 2, 4 ], [ 6, 8 ] ] is not moved by any permutation from G_([ [ [ 6, 8 ] ] ]). Therefore H = G_([ [ [ 6, 8 ]
              ] ]), the group generated by (2,4) and (6,8).
            QED(pointwise stabilizer)
            Now extend each generator of the stabilizer to A_1.
            The extension of (6,8) is (6,8). Now we show that (6,8) is in A_1:
              The element a = (6,8) can be written as follows as a word in the generators of A_1: a = a_11, and so belongs to A_1.
            QED(isin)
            The extension of (6,8) to a permutation that leaves E_1 invariant is (6,8)
            Proof:
              The permutation (6,8)|_G_1 = (6,8) and (6,8) is an automorphism of G_1. This can all be easily verified.
            QED(GetFamilyPerm)
            The extension of (2,4) is (2,4). Now we show that (2,4) is in A_1:
              The element a = (2,4) can be written as follows as a word in the generators of A_1: a = a_12, and so belongs to A_1.
            QED(isin)
            The extension of (2,4) to a permutation that leaves E_1 invariant is (2,4)
            Proof:
              The permutation (2,4)|_G_1 = (2,4) and (2,4) is an automorphism of G_1. This can all be easily verified.
            QED(GetFamilyPerm)
          QED(Step 2)
            These are formed by the symmetric groups of the vertices in V_2 for each family.
          Step 3: Proposition: there are no automorphisms on V_2 that leav V_1 invariant.
          QED(Step 3)
          By combining these steps we find that A_2 is [ (6,8), (2,4) ]
        QED(A_1->A_2)
        Proposition: A_3 = [ (6,8), (2,4) ].
        Proof:
          We now calculate A_3 in three steps.
          Step 1: Proposition: the vertexwise stabilizer (H) of A_2(=A) on V_2 is the group generated by (6,8) and (2,4).
            Proof:
              It is easy to see that the point 3 is not moved by any permutation from A. Therefore A_([ 3 ]) = A, the group generated by (6,8) and (2,4).
              It is easy to see that the point 7 is not moved by any permutation from A_([ 3 ]). Therefore H = A_([ 3 ]), the group generated by (6,8) and
              (2,4).
            QED(pointwise stabilizer)
          QED(Step 1)
          Step 2: Proposition: all automorphisms leave V_2 invariant.
            The reduction of A_2 to V_2 is [ () ]
            It is easy to verify that these are the cycles of permutations in A_2 that contain only vertices in V_2 (note that a cycle containing one vertex
            of V_2 must contain only vertices in V_2).
            Now we calculate the families.
              The families between V_2 = [ 3, 7 ] and V_3 = [ ], that are connected by edges E_2 = [ ] are:
               For this to be the set of families it is necessary that each vertex of V_3 occurs exactly once, the vertices in a family from V_3 are connected
              to those in that family from V_2 and these connections form all the edges in E_2. This can all be easily checked.
            QED(FamilyBlocks)
            Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
            of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
              Since G is the trivial group <()>, the pointwise stabilizer H of [ ] must be the trivial group <()>.

            QED(pointwise stabilizer)
            Now extend each generator of the stabilizer to A_2.
            The extension of () is (). Now we show that () is in A_2:
              The element a = () can be written as follows as a word in the generators of A_2: a = (), and so belongs to A_2.
            QED(isin)
          QED(Step 2)
          Step 3: V_3 is empty, so there are no nontrivial automorphisms to add here.
          QED(Step 3)
          By combining these steps we find that A_3 is [ (6,8), (2,4) ]
        QED(A_2->A_3)
      QED(GraphAutomorphism)
    QED(case v = 1)
    case v = 2
      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 2
      of H with a new edge.
      The resulting graph F has vertices [ 1 .. 8 ], edges [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8
      ] ] and new coloring [ 1 6 | 2:5 7 8 ].
      We now calculate the automorphism group of F and check whether there exists an automorphism that transforms 1 to 6.
        First we split V and E depending on the distance to e. The proof as done is by checking.
        Proposition: on the graph with vertices V=[1..8] and edges E=[ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8
        ], [ 7, 8 ] ], the edge e=[ 1, 6 ] splits the vertices and edges thus:
          V is split in V_i, sets of vertices at distance i to e:
```

```
  V_0 = [ 1, 6 ],
  V_1 = [ 2, 4, 5, 7, 8 ] and
  V_2 = [ 3 ].
  E is split in E_i, sets of edges between vertices at distance i to e and vertices of distance i or i+1 to e:
  E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 5, 6 ], [ 6, 7 ], [ 6, 8 ] ] and
  E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 5, 8 ], [ 7, 8 ] ].
Proof:
  It suffices to verify that (i) the V_i and E_i are disjoint, (ii) the union of the V_i is V, (iii) the union of the E_i is E, (iv) the edges in E_i
  are between a vertex in V_i and one in V_i or V_i+1 and (v) V_i is at distance at most i to e.
QED(VertexSets)
A_0 is [ (1,6) ].
Proposition: A_1 = [ (5,7), (5,7,8), (2,4) ].
Proof:
  We now calculate A_1 in three steps.
  Step 1: Proposition: the vertexwise stabilizer (H) of A_0(=A) on V_0 is the group generated by ().
    Proof:
      Proposition: Let A be the group generated by a1 = (1,6). Let A_([ 1 ]) be the trivial group <()>. Then the stabilizer of x = 1 in A is A_([ 1 ]).
      Proof:
        First we check that A_([ 1 ]) is a subgroup of A:
        Proposition: Let A_([ 1 ]) be the trivial group <()>. Let A be the group generated by a1 = (1,6). Then H is a subgroup of G.
        Proof:
          It suffices to verify that each of the generators of A_([ 1 ]) belongs to A. Below we give a word in the generators of A for each generator
          of A_([ 1 ]).
          Proof:
          QED(proof by generator)
          This establishes that each generator of A_([ 1 ]) belongs to A, and so A_([ 1 ]) is a subgroup of A.
        QED(issubgroup)
        Next we determine the Schreier data for A at x:
        Proposition: the Schreier data for A, the group generated by a1 = (1,6), at x = 1 is the table
          [ 1, 6 ]
          [ 0, 1 ]
          [ 0, 1 ]
          It has three rows, the first of which represents the A-orbit X of 1.
          Proposition: the orbit of x = 1 under A, the group generated by a1 = (1,6), is X = [ 1, 6 ].
          Proof:
            By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A
            leaves X invariant. It remains to show that the points in X are images of x under elements of A:
              1 = 1^()
              6 = 1^(1,6) = 1^a1
          QED(orbit of x = 1)
          To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
          verify that the image of x under the (-v)-th generator of A equals z if v<0, and that the image of x under the inverse of the v-the
          generator equals z if v>0.
           1 () = 0
           6 (1,6) = 1
          To finish, note that each z occurs earlier than x in the orbit list.
        QED(schreierdata)
        From the Schreier data we find that the Schreier elements are as follows:
        t(1) =  ()  = ()
        t(6) =  a1^-1  = (1,6)
        Finally we check that each Schreier generator is in A_([ 1 ]):
        Schreier gen(1,1)  = a1 t(a1)^(-1)  = () = ()
        Schreier gen(6,1)  = () t(())^(-1)  = a1^2 = ()
        By Schreier's lemma, A_([ 1 ]) is the stabilizer in A of x.
      QED(stabilizer)
      It is easy to see that the point 6 is not moved by any permutation from A_([ 1 ]). Therefore H = A_([ 1 ]), the group generated by ().
    QED(pointwise stabilizer)
  QED(Step 1)
  Step 2: Proposition: all automorphisms leave V_0 invariant.
    The reduction of A_0 to V_0 is [ (1,6) ]
    It is easy to verify that these are the cycles of permutations in A_0 that contain only vertices in V_0 (note that a cycle containing one vertex
    of V_0 must contain only vertices in V_0).
    Now we calculate the families.
      The families between V_0 = [ 1, 6 ] and V_1 = [ 2, 4, 5, 7, 8 ], that are connected by edges E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 5, 6 ], [
      6, 7 ], [ 6, 8 ] ] are:
      The families of type (2, 0) are [ [ [ 1, 6 ], [ ] ] ].
      The families of type (1, 2) are [ [ [ 1 ], [ 2, 4 ] ] ].
      The families of type (1, 3) are [ [ [ 6 ], [ 5, 7, 8 ] ] ].
       For this to be the set of families it is necessary that each vertex of V_1 occurs exactly once, the vertices in a family from V_1 are connected
      to those in that family from V_0 and these connections form all the edges in E_0. This can all be easily checked.
    QED(FamilyBlocks)
    Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
    of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
    Proof:
      It is easy to see that the point [ [ 1, 6 ] ] is not moved by any permutation from G. Therefore G_([ [ [ 1, 6 ] ] ]) = G, the group generated by
      (1,6).
      Proposition: Let G_([ [ [ 1, 6 ] ] ]) be the group generated by g_([ [ [ 1, 6 ] ] ])1 = (1,6). Let G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) be the
      trivial group <()>. Then the stabilizer of x = [ [ 1 ] ] in G_([ [ [ 1, 6 ] ] ]) is G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]).
      Proof:
        First we check that G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) is a subgroup of G_([ [ [ 1, 6 ] ] ]):
        Proposition: Let G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) be the trivial group <()>. Let G_([ [ [ 1, 6 ] ] ]) be the group generated by g_([ [ [ 1, 6 ]
        ] ])1 = (1,6). Then H is a subgroup of G.
        Proof:
          It suffices to verify that each of the generators of G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) belongs to G_([ [ [ 1, 6 ] ] ]). Below we give a word
          in the generators of G_([ [ [ 1, 6 ] ] ]) for each generator of G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]).
          Proof:
          QED(proof by generator)
          This establishes that each generator of G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) belongs to G_([ [ [ 1, 6 ] ] ]), and so G_([ [ [ 1, 6 ] ], [ [ 1 ] ]
          ]) is a subgroup of G_([ [ [ 1, 6 ] ] ]).
        QED(issubgroup)
        Next we determine the Schreier data for G_([ [ [ 1, 6 ] ] ]) at x:
        Proposition: the Schreier data for G_([ [ [ 1, 6 ] ] ]), the group generated by g_([ [ [ 1, 6 ] ] ])1 = (1,6), at x = [ [ 1 ] ] is the table
          [ [ [ 1 ] ], [ [ 6 ] ] ]
          [ 0, 1 ]
          [ [ [ 0 ] ], [ [ 1 ] ] ]
          It has three rows, the first of which represents the G_([ [ [ 1, 6 ] ] ])-orbit X of [ [ 1 ] ].
          Proposition: the orbit of x = [ [ 1 ] ] under G_([ [ [ 1, 6 ] ] ]), the group generated by g_([ [ [ 1, 6 ] ] ])1 = (1,6), is X = [ [ [ 1 ]
          ], [ [ 6 ] ] ].
          Proof:
            By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of G_([
            [ [ 1, 6 ] ] ]) leaves X invariant. It remains to show that the points in X are images of x under elements of G_([ [ [ 1, 6 ] ] ]):
              [ [ 1 ] ] = [ [ 1 ] ]^()
              [ [ 6 ] ] = [ [ 1 ] ]^(1,6) = [ [ 1 ] ]^g_([ [ [ 1, 6 ] ] ])1
          QED(orbit of x = [ [ 1 ] ])
          To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
          verify that the image of x under the (-v)-th generator of G_([ [ [ 1, 6 ] ] ]) equals z if v<0, and that the image of x under the inverse of
```

```
                the v-the generator equals z if v>0.
                 [ [ 1 ] ] () = [ [ 0 ] ]
                 [ [ 6 ] ] (1,6) = [ [ 1 ] ]
                To finish, note that each z occurs earlier than x in the orbit list.
              QED(schreierdata)
              From the Schreier data we find that the Schreier elements are as follows:
              t([ [ 1 ] ]) =  ()  = ()
              t([ [ 6 ] ]) =  g_([ [ [ 1, 6 ] ] ])1^-1  = (1,6)
              Finally we check that each Schreier generator is in G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]):
              Schreier gen([ [ 1 ] ],1)  = g_([ [ [ 1, 6 ] ] ])1 t(g_([ [ [ 1, 6 ] ] ])1)^(-1)  = () = ()
              Schreier gen([ [ 6 ] ],1)  = () t(())^(-1)  = g_([ [ [ 1, 6 ] ] ])1^2 = ()
              By Schreier's lemma, G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]) is the stabilizer in G_([ [ [ 1, 6 ] ] ]) of x.
            QED(stabilizer)
            It is easy to see that the point [ [ 6 ] ] is not moved by any permutation from G_([ [ [ 1, 6 ] ], [ [ 1 ] ] ]). Therefore H = G_([ [ [ 1, 6 ]
            ], [ [ 1 ] ] ]), the group generated by ().
          QED(pointwise stabilizer)
          Now extend each generator of the stabilizer to A_0.
        QED(Step 2)
          These are formed by the symmetric groups of the vertices in V_1 for each family.
        Step 3: Proposition: the automorphisms on V_1 that leave V_0 invariant are (2,4), (5,7,8) and (5,7).
          The family [ [ 1 ], [ 2, 4 ] ] adds the generators [ (2,4) ].
          The family [ [ 6 ], [ 5, 7, 8 ] ] adds the generators [ (5,7,8), (5,7) ].
        QED(Step 3)
        By combining these steps we find that A_1 is [ (5,7), (5,7,8), (2,4) ]
      QED(A_0->A_1)
      Proposition: A_2 = [ (5,7), (2,4) ].
      Proof:
        We now calculate A_2 in three steps.
        Step 1: Proposition: the vertexwise stabilizer (H) of A_1(=A) on V_1 is the group generated by ().
          Proof:
            Proposition: Let A be the group generated by a1 = (5,7), a2 = (5,7,8) and a3 = (2,4). Let A_([ 2 ]) be the group generated by a_([ 2 ])1 = (5,7)
            and a_([ 2 ])2 = (5,7,8). Then the stabilizer of x = 2 in A is A_([ 2 ]).
            Proof:
              First we check that A_([ 2 ]) is a subgroup of A:
              Proposition: Let A_([ 2 ]) be the group generated by a_([ 2 ])1 = (5,7) and a_([ 2 ])2 = (5,7,8). Let A be the group generated by a1 = (5,7),
              a2 = (5,7,8) and a3 = (2,4). Then H is a subgroup of G.
              Proof:
                It suffices to verify that each of the generators of A_([ 2 ]) belongs to A. Below we give a word in the generators of A for each generator
                of A_([ 2 ]).
                Proof:
                  a_([ 2 ])1 = (5,7) = a1;
                  a_([ 2 ])2 = (5,7,8) = a2.
                QED(proof by generator)
                This establishes that each generator of A_([ 2 ]) belongs to A, and so A_([ 2 ]) is a subgroup of A.
              QED(issubgroup)
              Next we determine the Schreier data for A at x:
              Proposition: the Schreier data for A, the group generated by a1 = (5,7), a2 = (5,7,8) and a3 = (2,4), at x = 2 is the table
                [ 2, 4 ]
                [ 0, 3 ]
                [ 0, 2 ]
                It has three rows, the first of which represents the A-orbit X of 2.
                Proposition: the orbit of x = 2 under A, the group generated by a1 = (5,7), a2 = (5,7,8) and a3 = (2,4), is X = [ 2, 4 ].
                Proof:
                  By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A
                  leaves X invariant. It remains to show that the points in X are images of x under elements of A:
                    2 = 2^()
                    4 = 2^(2,4) = 2^a3
                QED(orbit of x = 2)
                To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
                verify that the image of x under the (-v)-th generator of A equals z if v<0, and that the image of x under the inverse of the v-the
                generator equals z if v>0.
                 2 () = 0
                 4 (2,4) = 2
                To finish, note that each z occurs earlier than x in the orbit list.
              QED(schreierdata)
              From the Schreier data we find that the Schreier elements are as follows:
              t(2) =  ()  = ()
              t(4) =  a3^-1  = (2,4)
              Finally we check that each Schreier generator is in A_([ 2 ]):
              Schreier gen(2,1)  = a1 t(a1)^(-1)  = a1 = a_([ 2 ])1
              Schreier gen(2,2)  = a2 t(a2)^(-1)  = a2 = a_([ 2 ])2
              Schreier gen(2,3)  = a3 t(a3)^(-1)  = () = ()
              Schreier gen(4,1)  = a3^-1*a1 t(a3^-1*a1)^(-1)  = a3*a1*a3^-1 = a_([ 2 ])1
              Schreier gen(4,2)  = a3^-1*a2 t(a3^-1*a2)^(-1)  = a3*a2*a3^-1 = a_([ 2 ])2
              Schreier gen(4,3)  = () t(())^(-1)  = a3^2 = ()
              By Schreier's lemma, A_([ 2 ]) is the stabilizer in A of x.
            QED(stabilizer)
            It is easy to see that the point 4 is not moved by any permutation from A_([ 2 ]). Therefore A_([ 2, 4 ]) = A_([ 2 ]), the group generated by
            (5,7) and (5,7,8).
            Proposition: Let A_([ 2, 4 ]) be the group generated by a_([ 2, 4 ])1 = (5,7) and a_([ 2, 4 ])2 = (5,7,8). Let A_([ 2, 4, 5 ]) be the group
            generated by a_([ 2, 4, 5 ])1 = (7,8). Then the stabilizer of x = 5 in A_([ 2, 4 ]) is A_([ 2, 4, 5 ]).
            Proof:
              First we check that A_([ 2, 4, 5 ]) is a subgroup of A_([ 2, 4 ]):
              Proposition: Let A_([ 2, 4, 5 ]) be the group generated by a_([ 2, 4, 5 ])1 = (7,8). Let A_([ 2, 4 ]) be the group generated by a_([ 2, 4 ])1
              = (5,7) and a_([ 2, 4 ])2 = (5,7,8). Then H is a subgroup of G.
              Proof:
                It suffices to verify that each of the generators of A_([ 2, 4, 5 ]) belongs to A_([ 2, 4 ]). Below we give a word in the generators of A_([
                2, 4 ]) for each generator of A_([ 2, 4, 5 ]).
                Proof:
                  a_([ 2, 4, 5 ])1 = (7,8) = a_([ 2, 4 ])1*a_([ 2, 4 ])2^2.
                QED(proof by generator)
                This establishes that each generator of A_([ 2, 4, 5 ]) belongs to A_([ 2, 4 ]), and so A_([ 2, 4, 5 ]) is a subgroup of A_([ 2, 4 ]).
              QED(issubgroup)
              Next we determine the Schreier data for A_([ 2, 4 ]) at x:
              Proposition: the Schreier data for A_([ 2, 4 ]), the group generated by a_([ 2, 4 ])1 = (5,7) and a_([ 2, 4 ])2 = (5,7,8), at x = 5 is the
              table
                [ 5, 7, 8 ]
                [ 0, 1, -2 ]
                [ 0, 5, 5 ]
                It has three rows, the first of which represents the A_([ 2, 4 ])-orbit X of 5.
                Proposition: the orbit of x = 5 under A_([ 2, 4 ]), the group generated by a_([ 2, 4 ])1 = (5,7) and a_([ 2, 4 ])2 = (5,7,8), is X = [ 5, 7,
                8 ].
                Proof:
                  By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A_([
                  2, 4 ]) leaves X invariant. It remains to show that the points in X are images of x under elements of A_([ 2, 4 ]):
                    5 = 5^()
```

```
                    7 = 5^(5,7) = 5^a_([ 2, 4 ])1
                    8 = 5^(5,8,7) = 5^a_([ 2, 4 ])2^2
                  QED(orbit of x = 5)
                  To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
590               verify that the image of x under the (-v)-th generator of A_([ 2, 4 ]) equals z if v<0, and that the image of x under the inverse of the
                  v-the generator equals z if v>0.
                   5 () = 0
                   7 (5,7) = 5
                   8 (5,7,8) = 5
595               To finish, note that each z occurs earlier than x in the orbit list.
                QED(schreierdata)
                From the Schreier data we find that the Schreier elements are as follows:
                t(5) =  ()  = ()
                t(7) =  a_([ 2, 4 ])1^-1  = (5,7)
600             t(8) =  a_([ 2, 4 ])2  = (5,7,8)
                Finally we check that each Schreier generator is in A_([ 2, 4, 5 ]):
                Schreier gen(5,1)  = a_([ 2, 4 ])1 t(a_([ 2, 4 ])1)^(-1)  = () = ()
                Schreier gen(5,2)  = a_([ 2, 4 ])2 t(a_([ 2, 4 ])2)^(-1)  = a_([ 2, 4 ])2*a_([ 2, 4 ])1^-1 = a_([ 2, 4, 5 ])1
                Schreier gen(7,1)  = () t(())^(-1)  = a_([ 2, 4 ])1^2 = ()
605             Schreier gen(7,2)  = a_([ 2, 4 ])1^-1*a_([ 2, 4 ])2 t(a_([ 2, 4 ])1^-1*a_([ 2, 4 ])2)^(-1)  = a_([ 2, 4 ])1*a_([ 2, 4 ])2^2 = a_([ 2, 4, 5 ])1
                Schreier gen(8,1)  = a_([ 2, 4 ])2*a_([ 2, 4 ])1 t(a_([ 2, 4 ])2*a_([ 2, 4 ])1)^(-1)  = a_([ 2, 4 ])2^-1*a_([ 2, 4 ])1*a_([ 2, 4 ])2 = a_([ 2,
                4, 5 ])1
                Schreier gen(8,2)  = a_([ 2, 4 ])2^2 t(a_([ 2, 4 ])2^2)^(-1)  = () = ()
                By Schreier's lemma, A_([ 2, 4, 5 ]) is the stabilizer in A_([ 2, 4 ]) of x.
610           QED(stabilizer)
              Proposition: Let A_([ 2, 4, 5 ]) be the group generated by a_([ 2, 4, 5 ])1 = (7,8). Let A_([ 2, 4, 5, 7 ]) be the trivial group <()>. Then the
              stabilizer of x = 7 in A_([ 2, 4, 5 ]) is A_([ 2, 4, 5, 7 ]).
              Proof:
                First we check that A_([ 2, 4, 5, 7 ]) is a subgroup of A_([ 2, 4, 5 ]):
615             Proposition: Let A_([ 2, 4, 5, 7 ]) be the trivial group <()>. Let A_([ 2, 4, 5 ]) be the group generated by a_([ 2, 4, 5 ])1 = (7,8). Then H
                is a subgroup of G.
                Proof:
                  It suffices to verify that each of the generators of A_([ 2, 4, 5, 7 ]) belongs to A_([ 2, 4, 5 ]). Below we give a word in the generators
                  of A_([ 2, 4, 5 ]) for each generator of A_([ 2, 4, 5, 7 ]).
620               Proof:
                  QED(proof by generator)
                  This establishes that each generator of A_([ 2, 4, 5, 7 ]) belongs to A_([ 2, 4, 5 ]), and so A_([ 2, 4, 5, 7 ]) is a subgroup of A_([ 2, 4,
                  5 ]).
                QED(issubgroup)
625             Next we determine the Schreier data for A_([ 2, 4, 5 ]) at x:
                Proposition: the Schreier data for A_([ 2, 4, 5 ]), the group generated by a_([ 2, 4, 5 ])1 = (7,8), at x = 7 is the table
                  [ 7, 8 ]
                  [ 0, 1 ]
                  [ 0, 7 ]
630               It has three rows, the first of which represents the A_([ 2, 4, 5 ])-orbit X of 7.
                  Proposition: the orbit of x = 7 under A_([ 2, 4, 5 ]), the group generated by a_([ 2, 4, 5 ])1 = (7,8), is X = [ 7, 8 ].
                  Proof:
                    By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A_([
                    2, 4, 5 ]) leaves X invariant. It remains to show that the points in X are images of x under elements of A_([ 2, 4, 5 ]):
635                   7 = 7^()
                      8 = 7^(7,8) = 7^a_([ 2, 4, 5 ])1
                  QED(orbit of x = 7)
                  To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
                  verify that the image of x under the (-v)-th generator of A_([ 2, 4, 5 ]) equals z if v<0, and that the image of x under the inverse of the
640               v-the generator equals z if v>0.
                   7 () = 0
                   8 (7,8) = 7
                  To finish, note that each z occurs earlier than x in the orbit list.
                QED(schreierdata)
645             From the Schreier data we find that the Schreier elements are as follows:
                t(7) =  ()  = ()
                t(8) =  a_([ 2, 4, 5 ])1^-1  = (7,8)
                Finally we check that each Schreier generator is in A_([ 2, 4, 5, 7 ]):
                Schreier gen(7,1)  = a_([ 2, 4, 5 ])1 t(a_([ 2, 4, 5 ])1)^(-1)  = () = ()
650             Schreier gen(8,1)  = () t(())^(-1)  = a_([ 2, 4, 5 ])1^2 = ()
                By Schreier's lemma, A_([ 2, 4, 5, 7 ]) is the stabilizer in A_([ 2, 4, 5 ]) of x.
              QED(stabilizer)
              It is easy to see that the point 8 is not moved by any permutation from A_([ 2, 4, 5, 7 ]). Therefore H = A_([ 2, 4, 5, 7 ]), the group
              generated by ().
655         QED(pointwise stabilizer)
          QED(Step 1)
          Step 2: Proposition: the extensions of the automorphisms that don't leave V_1 invariant are (5,7) and (2,4)
            The reduction of A_1 to V_1 is [ (5,7), (5,7,8), (2,4) ]
            It is easy to verify that these are the cycles of permutations in A_1 that contain only vertices in V_1 (note that a cycle containing one vertex
660         of V_1 must contain only vertices in V_1).
            Now we calculate the families.
              The families between V_1 = [ 2, 4, 5, 7, 8 ] and V_2 = [ 3 ], that are connected by edges E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 5, 8 ], [ 7, 8 ] ] are:
              The families of type (2, 0) are [ [ [ 5, 8 ], [ ] ], [ [ 7, 8 ], [ ] ] ].
              The families of type (2, 1) are [ [ [ 2, 4 ], [ 3 ] ] ].
665            For this to be the set of families it is necessary that each vertex of V_2 occurs exactly once, the vertices in a family from V_2 are connected
              to those in that family from V_1 and these connections form all the edges in E_1. This can all be easily checked.
            QED(FamilyBlocks)
            Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
            of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
670         Proof:
              Proposition: Let G be the group generated by g1 = (5,7), g2 = (5,7,8) and g3 = (2,4). Let G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) be the group generated
              by g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1 = (2,4) and g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])2 = (5,7). Then the stabilizer of x = [ [ 5, 8 ], [ 7, 8 ] ] in G
              is G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]).
              Proof:
675             First we check that G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) is a subgroup of G:
                Proposition: Let G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) be the group generated by g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1 = (2,4) and g_([ [ [ 5, 8 ], [ 7, 8
                ] ] ])2 = (5,7). Let G be the group generated by g1 = (5,7), g2 = (5,7,8) and g3 = (2,4). Then H is a subgroup of G.
                Proof:
                  It suffices to verify that each of the generators of G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) belongs to G. Below we give a word in the generators of
680               G for each generator of G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]).
                  Proof:
                    g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1 = (2,4) = g3;
                    g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])2 = (5,7) = g1.
                  QED(proof by generator)
685               This establishes that each generator of G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) belongs to G, and so G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) is a subgroup of
                  G.
                QED(issubgroup)
                Next we determine the Schreier data for G at x:
                Proposition: the Schreier data for G, the group generated by g1 = (5,7), g2 = (5,7,8) and g3 = (2,4), at x = [ [ 5, 8 ], [ 7, 8 ] ] is the
690             table
                  [ [ [ 5, 8 ], [ 7, 8 ] ], [ [ 5, 7 ], [ 5, 8 ] ], [ [ 5, 7 ], [ 7, 8 ] ] ]
```

```
                    [ 0, 2, -2 ]
                    [ [ [ 0, 0 ], [ 0, 0 ] ], [ [ 5, 8 ], [ 7, 8 ] ], [ [ 5, 8 ], [ 7, 8 ] ] ]
                    It has three rows, the first of which represents the G-orbit X of [ [ 5, 8 ], [ 7, 8 ] ].
695                 Proposition: the orbit of x = [ [ 5, 8 ], [ 7, 8 ] ] under G, the group generated by g1 = (5,7), g2 = (5,7,8) and g3 = (2,4), is X = [ [ [
                    5, 7 ], [ 5, 8 ] ], [ [ 5, 7 ], [ 7, 8 ] ], [ [ 5, 8 ], [ 7, 8 ] ] ].
                    Proof:
                      By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of G
                      leaves X invariant. It remains to show that the points in X are images of x under elements of G:
700                     [ [ 5, 7 ], [ 5, 8 ] ] = [ [ 5, 8 ], [ 7, 8 ] ]^(5,7,8) = [ [ 5, 8 ], [ 7, 8 ] ]^g2
                        [ [ 5, 7 ], [ 7, 8 ] ] = [ [ 5, 8 ], [ 7, 8 ] ]^(5,8,7) = [ [ 5, 8 ], [ 7, 8 ] ]^g2^2
                        [ [ 5, 8 ], [ 7, 8 ] ] = [ [ 5, 8 ], [ 7, 8 ] ]^()
                    QED(orbit of x = [ [ 5, 8 ], [ 7, 8 ] ])
                    To show that the second row is a Schreier vector and the third row are its backpointers, we consider each column x,v,z of the table and
705                 verify that the image of x under the (-v)-th generator of G equals z if v<0, and that the image of x under the inverse of the v-the
                    generator equals z if v>0.
                     [ [ 5, 8 ], [ 7, 8 ] ] () = [ [ 0, 0 ], [ 0, 0 ] ]
                     [ [ 5, 7 ], [ 5, 8 ] ] (5,8,7) = [ [ 5, 8 ], [ 7, 8 ] ]
                     [ [ 5, 7 ], [ 7, 8 ] ] (5,7,8) = [ [ 5, 8 ], [ 7, 8 ] ]
710                 To finish, note that each z occurs earlier than x in the orbit list.
                  QED(schreierdata)
                  From the Schreier data we find that the Schreier elements are as follows:
                  t([ [ 5, 8 ], [ 7, 8 ] ]) =  ()  = ()
                  t([ [ 5, 7 ], [ 5, 8 ] ]) =  g2^-1  = (5,8,7)
715               t([ [ 5, 7 ], [ 7, 8 ] ]) =  g2  = (5,7,8)
                  Finally we check that each Schreier generator is in G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]):
                  Schreier gen([ [ 5, 8 ], [ 7, 8 ] ],1)  = g1 t(g1)^(-1)  = g1 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])2
                  Schreier gen([ [ 5, 8 ], [ 7, 8 ] ],2)  = g2 t(g2)^(-1)  = () = ()
                  Schreier gen([ [ 5, 8 ], [ 7, 8 ] ],3)  = g3 t(g3)^(-1)  = g3 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1
720               Schreier gen([ [ 5, 7 ], [ 5, 8 ] ],1)  = g2^-1*g1 t(g2^-1*g1)^(-1)  = g2*g1*g2 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])2
                  Schreier gen([ [ 5, 7 ], [ 5, 8 ] ],2)  = () t(())^(-1)  = g2^3 = ()
                  Schreier gen([ [ 5, 7 ], [ 5, 8 ] ],3)  = g2^-1*g3 t(g2^-1*g3)^(-1)  = g2*g3*g2^-1 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1
                  Schreier gen([ [ 5, 7 ], [ 7, 8 ] ],1)  = g2*g1 t(g2*g1)^(-1)  = g2^-1*g1*g2^-1 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])2
                  Schreier gen([ [ 5, 7 ], [ 7, 8 ] ],2)  = g2^2 t(g2^2)^(-1)  = () = ()
725               Schreier gen([ [ 5, 7 ], [ 7, 8 ] ],3)  = g2*g3 t(g2*g3)^(-1)  = g2^-1*g3*g2 = g_([ [ [ 5, 8 ], [ 7, 8 ] ] ])1
                  By Schreier's lemma, G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]) is the stabilizer in G of x.
                QED(stabilizer)
                It is easy to see that the point [ [ 2, 4 ] ] is not moved by any permutation from G_([ [ [ 5, 8 ], [ 7, 8 ] ] ]). Therefore H = G_([ [ [ 5, 8
                ], [ 7, 8 ] ] ]), the group generated by (2,4) and (5,7).
730           QED(pointwise stabilizer)
              Now extend each generator of the stabilizer to A_1.
              The extension of (5,7) is (5,7). Now we show that (5,7) is in A_1:
                The element a = (5,7) can be written as follows as a word in the generators of A_1: a = a_11, and so belongs to A_1.
              QED(isin)
735           The extension of (5,7) to a permutation that leaves E_1 invariant is (5,7)
              Proof:
                The permutation (5,7)|_G_1 = (5,7) and (5,7) is an automorphism of G_1. This can all be easily verified.
              QED(GetFamilyPerm)
              The extension of (2,4) is (2,4). Now we show that (2,4) is in A_1:
740             The element a = (2,4) can be written as follows as a word in the generators of A_1: a = a_13, and so belongs to A_1.
              QED(isin)
              The extension of (2,4) to a permutation that leaves E_1 invariant is (2,4)
              Proof:
                The permutation (2,4)|_G_1 = (2,4) and (2,4) is an automorphism of G_1. This can all be easily verified.
745           QED(GetFamilyPerm)
            QED(Step 2)
              These are formed by the symmetric groups of the vertices in V_2 for each family.
            Step 3: Proposition: there are no automorphisms on V_2 that leav V_1 invariant.
            QED(Step 3)
750         By combining these steps we find that A_2 is [ (5,7), (2,4) ]
          QED(A_1->A_2)
          Proposition: A_3 = [ (5,7), (2,4) ].
          Proof:
            We now calculate A_3 in three steps.
755         Step 1: Proposition: the vertexwise stabilizer (H) of A_2(=A) on V_2 is the group generated by (5,7) and (2,4).
              Proof:
                It is easy to see that the point 3 is not moved by any permutation from A. Therefore H = A, the group generated by (5,7) and (2,4).
              QED(pointwise stabilizer)
            QED(Step 1)
760         Step 2: Proposition: all automorphisms leave V_2 invariant.
              The reduction of A_2 to V_2 is [ () ]
              It is easy to verify that these are the cycles of permutations in A_2 that contain only vertices in V_2 (note that a cycle containing one vertex
              of V_2 must contain only vertices in V_2).
              Now we calculate the families.
765             The families between V_2 = [ 3 ] and V_3 = [ ], that are connected by edges E_2 = [ ] are:
                 For this to be the set of families it is necessary that each vertex of V_3 occurs exactly once, the vertices in a family from V_3 are connected
                to those in that family from V_2 and these connections form all the edges in E_2. This can all be easily checked.
              QED(FamilyBlocks)
              Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
770           of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
                Since G is the trivial group <()>, the pointwise stabilizer H of [ ] must be the trivial group <()>.

              QED(pointwise stabilizer)
              Now extend each generator of the stabilizer to A_2.
775           The extension of () is (). Now we show that () is in A_2:
                The element a = () can be written as follows as a word in the generators of A_2: a = (), and so belongs to A_2.
              QED(isin)
            QED(Step 2)
            Step 3: V_3 is empty, so there are no nontrivial automorphisms to add here.
780         QED(Step 3)
            By combining these steps we find that A_3 is [ (5,7), (2,4) ]
          QED(A_2->A_3)
        QED(GraphAutomorphism)
      QED(case v = 2)
785   QED(case distinction)
QED(graphisomorphism)
```

Short version without full orbit and stabilizer proofs

This is the short version of the proof, where orbits and stabilizers proofs are omitted.

```
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 ].
  An orbit proof was hidden.
QED(all orbits)
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.
      First we split V and E depending on the distance to e. The proof as done is by checking.
      Proposition: on the graph with vertices V=[1..8] and edges E=[ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8
      ], [ 7, 8 ] ], the edge e=[ 1, 5 ] splits the vertices and edges thus:
        V is split in V_i, sets of vertices at distance i to e:
        V_0 = [ 1, 5 ],
        V_1 = [ 2, 4, 6, 8 ] and
        V_2 = [ 3, 7 ].
        E is split in E_i, sets of edges between vertices at distance i to e and vertices of distance i or i+1 to e:
        E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 5, 6 ], [ 5, 8 ] ] and
        E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8 ] ].
      Proof:
        It suffices to verify that (i) the V_i and E_i are disjoint, (ii) the union of the V_i is V, (iii) the union of the E_i is E, (iv) the edges in E_i
        are between a vertex in V_i and one in V_i or V_i+1 and (v) V_i is at distance at most i to e.
      QED(VertexSets)
      A_0 is [ (1,5) ].
      Proposition: A_1 = [ (6,8), (2,4), (1,5)(2,6)(4,8) ].
      Proof:
        We now calculate A_1 in three steps.
        Step 1: Proposition: the vertexwise stabilizer (H) of A_0(=A) on V_0 is the group generated by ().
            A stabilizer proof was hidden.
          QED(pointwise stabilizer)
        QED(Step 1)
        Step 2: Proposition: the extension of the automorphism that doesn't leave V_0 invariant is (1,5)(2,6)(4,8)
          The reduction of A_0 to V_0 is [ (1,5) ]
          It is easy to verify that these are the cycles of permutations in A_0 that contain only vertices in V_0 (note that a cycle containing one vertex
          of V_0 must contain only vertices in V_0).
          Now we calculate the families.
            The families between V_0 = [ 1, 5 ] and V_1 = [ 2, 4, 6, 8 ], that are connected by edges E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 5 ], [ 5, 6 ], [ 5, 8
            ] ] are:
            The families of type (2, 0) are [ [ [ 1, 5 ], [ ] ] ].
            The families of type (1, 2) are [ [ [ 1 ], [ 2, 4 ] ], [ [ 5 ], [ 6, 8 ] ] ].
             For this to be the set of families it is necessary that each vertex of V_1 occurs exactly once, the vertices in a family from V_1 are connected
            to those in that family from V_0 and these connections form all the edges in E_0. This can all be easily checked.
          QED(FamilyBlocks)
          Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
          of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
            A stabilizer proof was hidden.
          QED(pointwise stabilizer)
          Now extend each generator of the stabilizer to A_0.
          The extension of (1,5) is (1,5). Now we show that (1,5) is in A_0:
            The element a = (1,5) can be written as follows as a word in the generators of A_0: a = a_01, and so belongs to A_0.
          QED(isin)
          The extension of (1,5) to a permutation that leaves E_0 invariant is (1,5)(2,6)(4,8)
          Proof:
            The permutation (1,5)(2,6)(4,8)|_G_0 = (1,5) and (1,5)(2,6)(4,8) is an automorphism of G_0. This can all be easily verified.
          QED(GetFamilyPerm)
        QED(Step 2)
          These are formed by the symmetric groups of the vertices in V_1 for each family.
        Step 3: Proposition: the automorphisms on V_1 that leave V_0 invariant are (2,4) and (6,8).
          The family [ [ 1 ], [ 2, 4 ] ] adds the generators [ (2,4) ].
          The family [ [ 5 ], [ 6, 8 ] ] adds the generators [ (6,8) ].
        QED(Step 3)
        By combining these steps we find that A_1 is [ (6,8), (2,4), (1,5)(2,6)(4,8) ]
      QED(A_0->A_1)
      Proposition: A_2 = [ (6,8), (2,4) ].
      Proof:
        We now calculate A_2 in three steps.
        Step 1: Proposition: the vertexwise stabilizer (H) of A_1(=A) on V_1 is the group generated by ().
            A stabilizer proof was hidden.
          QED(pointwise stabilizer)
        QED(Step 1)
        Step 2: Proposition: the extensions of the automorphisms that don't leave V_1 invariant are (6,8) and (2,4)
          The reduction of A_1 to V_1 is [ (6,8), (2,4), (2,6)(4,8) ]
          It is easy to verify that these are the cycles of permutations in A_1 that contain only vertices in V_1 (note that a cycle containing one vertex
          of V_1 must contain only vertices in V_1).
          Now we calculate the families.
            The families between V_1 = [ 2, 4, 6, 8 ] and V_2 = [ 3, 7 ], that are connected by edges E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8
            ] ] are:
            The families of type (2, 0) are [ [ [ 6, 8 ], [ ] ] ].
            The families of type (2, 1) are [ [ [ 2, 4 ], [ 3 ] ], [ [ 6, 8 ], [ 7 ] ] ].
             For this to be the set of families it is necessary that each vertex of V_2 occurs exactly once, the vertices in a family from V_2 are connected
            to those in that family from V_1 and these connections form all the edges in E_1. This can all be easily checked.
          QED(FamilyBlocks)
          Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
          of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
            A stabilizer proof was hidden.
          QED(pointwise stabilizer)
          Now extend each generator of the stabilizer to A_1.
          The extension of (6,8) is (6,8). Now we show that (6,8) is in A_1:
            The element a = (6,8) can be written as follows as a word in the generators of A_1: a = a_11, and so belongs to A_1.
          QED(isin)
          The extension of (6,8) to a permutation that leaves E_1 invariant is (6,8)
          Proof:
            The permutation (6,8)|_G_1 = (6,8) and (6,8) is an automorphism of G_1. This can all be easily verified.
          QED(GetFamilyPerm)
          The extension of (2,4) is (2,4). Now we show that (2,4) is in A_1:
            The element a = (2,4) can be written as follows as a word in the generators of A_1: a = a_12, and so belongs to A_1.
          QED(isin)
          The extension of (2,4) to a permutation that leaves E_1 invariant is (2,4)
          Proof:
            The permutation (2,4)|_G_1 = (2,4) and (2,4) is an automorphism of G_1. This can all be easily verified.
          QED(GetFamilyPerm)
```

```
      QED(Step 2)
        These are formed by the symmetric groups of the vertices in V_2 for each family.
      Step 3: Proposition: there are no automorphisms on V_2 that leav V_1 invariant.
      QED(Step 3)
      By combining these steps we find that A_2 is [ (6,8), (2,4) ]
    QED(A_1->A_2)
    Proposition: A_3 = [ (6,8), (2,4) ].
    Proof:
      We now calculate A_3 in three steps.
      Step 1: Proposition: the vertexwise stabilizer (H) of A_2(=A) on V_2 is the group generated by (6,8) and (2,4).
          A stabilizer proof was hidden.
        QED(pointwise stabilizer)
      QED(Step 1)
      Step 2: Proposition: all automorphisms leave V_2 invariant.
        The reduction of A_2 to V_2 is [ () ]
        It is easy to verify that these are the cycles of permutations in A_2 that contain only vertices in V_2 (note that a cycle containing one vertex
        of V_2 must contain only vertices in V_2).
        Now we calculate the families.
          The families between V_2 = [ 3, 7 ] and V_3 = [ ], that are connected by edges E_2 = [ ] are:
           For this to be the set of families it is necessary that each vertex of V_3 occurs exactly once, the vertices in a family from V_3 are connected
          to those in that family from V_2 and these connections form all the edges in E_2. This can all be easily checked.
        QED(FamilyBlocks)
        Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
        of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
          A stabilizer proof was hidden.
        QED(pointwise stabilizer)
        Now extend each generator of the stabilizer to A_2.
        The extension of () is (). Now we show that () is in A_2:
          The element a = () can be written as follows as a word in the generators of A_2: a = (), and so belongs to A_2.
        QED(isin)
      QED(Step 2)
      Step 3: V_3 is empty, so there are no nontrivial automorphisms to add here.
      QED(Step 3)
      By combining these steps we find that A_3 is [ (6,8), (2,4) ]
    QED(A_2->A_3)
  QED(GraphAutomorphism)
QED(case v = 1)
case v = 2
  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 2
  of H with a new edge.
  The resulting graph F has vertices [ 1 .. 8 ], edges [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8
  ] ] and new coloring [ 1 6 | 2:5 7 8 ].
  We now calculate the automorphism group of F and check whether there exists an automorphism that transforms 1 to 6.
    First we split V and E depending on the distance to e. The proof as done is by checking.
    Proposition: on the graph with vertices V=[1..8] and edges E=[ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8
    ], [ 7, 8 ] ], the edge e=[ 1, 6 ] splits the vertices and edges thus:
      V is split in V_i, sets of vertices at distance i to e:
      V_0 = [ 1, 6 ],
      V_1 = [ 2, 4, 5, 7, 8 ] and
      V_2 = [ 3 ].
      E is split in E_i, sets of edges between vertices at distance i to e and vertices of distance i or i+1 to e:
      E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 5, 6 ], [ 6, 7 ], [ 6, 8 ] ] and
      E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 5, 8 ], [ 7, 8 ] ].
    Proof:
      It suffices to verify that (i) the V_i and E_i are disjoint, (ii) the union of the V_i is V, (iii) the union of the E_i is E, (iv) the edges in E_i
      are between a vertex in V_i and one in V_i or V_i+1 and (v) V_i is at distance at most i to e.
    QED(VertexSets)
    A_0 is [ (1,6) ].
    Proposition: A_1 = [ (5,7), (5,7,8), (2,4) ].
    Proof:
      We now calculate A_1 in three steps.
      Step 1: Proposition: the vertexwise stabilizer (H) of A_0(=A) on V_0 is the group generated by ().
          A stabilizer proof was hidden.
        QED(pointwise stabilizer)
      QED(Step 1)
      Step 2: Proposition: all automorphisms leave V_0 invariant.
        The reduction of A_0 to V_0 is [ (1,6) ]
        It is easy to verify that these are the cycles of permutations in A_0 that contain only vertices in V_0 (note that a cycle containing one vertex
        of V_0 must contain only vertices in V_0).
        Now we calculate the families.
          The families between V_0 = [ 1, 6 ] and V_1 = [ 2, 4, 5, 7, 8 ], that are connected by edges E_0 = [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 5, 6 ], [
          6, 7 ], [ 6, 8 ] ] are:
          The families of type (2, 0) are [ [ [ 1, 6 ], [ ] ] ].
          The families of type (1, 2) are [ [ [ 1 ], [ 2, 4 ] ] ].
          The families of type (1, 3) are [ [ [ 6 ], [ 5, 7, 8 ] ] ].
           For this to be the set of families it is necessary that each vertex of V_1 occurs exactly once, the vertices in a family from V_1 are connected
          to those in that family from V_0 and these connections form all the edges in E_0. This can all be easily checked.
        QED(FamilyBlocks)
        Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
        of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
          A stabilizer proof was hidden.
        QED(pointwise stabilizer)
        Now extend each generator of the stabilizer to A_0.
      QED(Step 2)
        These are formed by the symmetric groups of the vertices in V_1 for each family.
      Step 3: Proposition: the automorphisms on V_1 that leave V_0 invariant are (2,4), (5,7,8) and (5,7).
        The family [ [ 1 ], [ 2, 4 ] ] adds the generators [ (2,4) ].
        The family [ [ 6 ], [ 5, 7, 8 ] ] adds the generators [ (5,7,8), (5,7) ].
      QED(Step 3)
      By combining these steps we find that A_1 is [ (5,7), (5,7,8), (2,4) ]
    QED(A_0->A_1)
    Proposition: A_2 = [ (5,7), (2,4) ].
    Proof:
      We now calculate A_2 in three steps.
      Step 1: Proposition: the vertexwise stabilizer (H) of A_1(=A) on V_1 is the group generated by ().
          A stabilizer proof was hidden.
        QED(pointwise stabilizer)
      QED(Step 1)
      Step 2: Proposition: the extensions of the automorphisms that don't leave V_1 invariant are (5,7) and (2,4)
        The reduction of A_1 to V_1 is [ (5,7), (5,7,8), (2,4) ]
        It is easy to verify that these are the cycles of permutations in A_1 that contain only vertices in V_1 (note that a cycle containing one vertex
        of V_1 must contain only vertices in V_1).
        Now we calculate the families.
          The families between V_1 = [ 2, 4, 5, 7, 8 ] and V_2 = [ 3 ], that are connected by edges E_1 = [ [ 2, 3 ], [ 3, 4 ], [ 5, 8 ], [ 7, 8 ] ] are:
          The families of type (2, 0) are [ [ [ 5, 8 ], [ ] ], [ [ 7, 8 ], [ ] ] ].
          The families of type (2, 1) are [ [ [ 2, 4 ], [ 3 ] ] ].
```

```
                  For this to be the set of families it is necessary that each vertex of V_2 occurs exactly once, the vertices in a family from V_2 are connected
                  to those in that family from V_1 and these connections form all the edges in E_1. This can all be easily checked.
                QED(FamilyBlocks)
220             Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
                of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
                  A stabilizer proof was hidden.
                QED(pointwise stabilizer)
                Now extend each generator of the stabilizer to A_1.
225             The extension of (5,7) is (5,7). Now we show that (5,7) is in A_1:
                  The element a = (5,7) can be written as follows as a word in the generators of A_1: a = a_11, and so belongs to A_1.
                QED(isin)
                The extension of (5,7) to a permutation that leaves E_1 invariant is (5,7)
                Proof:
230               The permutation (5,7)|_G_1 = (5,7) and (5,7) is an automorphism of G_1. This can all be easily verified.
                QED(GetFamilyPerm)
                The extension of (2,4) is (2,4). Now we show that (2,4) is in A_1:
                  The element a = (2,4) can be written as follows as a word in the generators of A_1: a = a_13, and so belongs to A_1.
                QED(isin)
235             The extension of (2,4) to a permutation that leaves E_1 invariant is (2,4)
                Proof:
                  The permutation (2,4)|_G_1 = (2,4) and (2,4) is an automorphism of G_1. This can all be easily verified.
                QED(GetFamilyPerm)
              QED(Step 2)
240             These are formed by the symmetric groups of the vertices in V_2 for each family.
              Step 3: Proposition: there are no automorphisms on V_2 that leav V_1 invariant.
              QED(Step 3)
              By combining these steps we find that A_2 is [ (5,7), (2,4) ]
            QED(A_1->A_2)
245         Proposition: A_3 = [ (5,7), (2,4) ].
            Proof:
              We now calculate A_3 in three steps.
              Step 1: Proposition: the vertexwise stabilizer (H) of A_2(=A) on V_2 is the group generated by (5,7) and (2,4).
                  A stabilizer proof was hidden.
250             QED(pointwise stabilizer)
              QED(Step 1)
              Step 2: Proposition: all automorphisms leave V_2 invariant.
                The reduction of A_2 to V_2 is [ () ]
                It is easy to verify that these are the cycles of permutations in A_2 that contain only vertices in V_2 (note that a cycle containing one vertex
255             of V_2 must contain only vertices in V_2).
                Now we calculate the families.
                  The families between V_2 = [ 3 ] and V_3 = [ ], that are connected by edges E_2 = [ ] are:
                   For this to be the set of families it is necessary that each vertex of V_3 occurs exactly once, the vertices in a family from V_3 are connected
                  to those in that family from V_2 and these connections form all the edges in E_2. This can all be easily checked.
260             QED(FamilyBlocks)
                Now we calculate the stabilizer (H) of the reduction (G) that leaves the list of sets of families invariant. Note that a family is a list of sets
                of vertices, so the action will be on tuples of sets of tuples of sets [add reference ?/need to change with color].
                  A stabilizer proof was hidden.
                QED(pointwise stabilizer)
265             Now extend each generator of the stabilizer to A_2.
                The extension of () is (). Now we show that () is in A_2:
                  The element a = () can be written as follows as a word in the generators of A_2: a = (), and so belongs to A_2.
                QED(isin)
              QED(Step 2)
270           Step 3: V_3 is empty, so there are no nontrivial automorphisms to add here.
              QED(Step 3)
              By combining these steps we find that A_3 is [ (5,7), (2,4) ]
            QED(A_2->A_3)
          QED(GraphAutomorphism)
275     QED(case v = 2)
      QED(case distinction)
    QED(graphisomorphism)
```

Chapter 4

McKay's algorithm

4.1 Introduction

McKay has developed an algorithm for graph isomorphism [27, 28] that is fast in practice but not known to be polynomial in time. The current implementation [26] is one of the most efficient practical graph isomorphism solvers available. We have modified this program to output a human-readable proof.

We chose not to use the algorithm that gives a canonical labeling and the methods implemented to reduce the tree by using invariants. It is harder to construct a proof for a canonical labeling than to construct a proof for the automorphism group of a graph. Furthermore the canonical labeling may change depending on some settings in the algorithm and therefore does not really give much information about the graph. The big disadvantage of using the automorphism group is that a new graph must be constructed from the two earlier graphs and that the resulting graphs is twice as big as the original graphs. We don't use the other invariants on colored graphs provided by nauty to reduce the tree because we would then have to describe and proof all of these invariants used. It might be possible to rewrite some of these to give enough output so that the result can be proved or recalculated easily.

4.2 Definitions and variables

Let $G = (V, E)$ be a graph. In Section 2.7, a *partition* of a graph was used to refer to a coloring of a graph with a finite ordered color set C that is onto. In this chapter it is also called a partition of the vertices (of the graph). A partition is denoted by sorting the vertices according to their color, 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 $u, v \in c$, $\mathrm{adj}_d(u) = \mathrm{adj}_d(v)$.

Let $\pi = [V_1 \mid \ldots \mid V_k]$ be a partition of V. Let $v \in V_i$, for some i. If $|V_i| > 1$ then $\pi \circ v$ is $(V_1, \ldots, V_{i-1}, \{v\}, V_i \setminus \{v\}, V_{i+1}, \ldots, V_k)$. [

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 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.

Algorithm 7 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
```

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 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

Algorithm 8 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
```

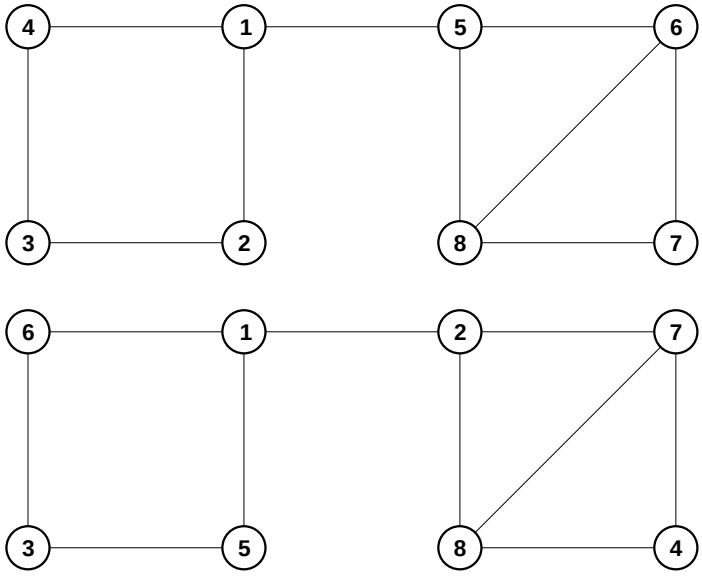

Figure 4.1: A graph and the corresponding new labeling resulting from the partition [1|5|3|7|2|4|6|8]

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 can not 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:

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 [27] and algorithm 2.5 in [28]). This is the standard algorithm that 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 is 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.

Algorithm 9 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
```

4.3 Implementation

We have chosen to determine the automorphism group of a graph rather than the canonical labeling, because the latter is dependent on the version of the nauty program and harder to understand. To check whether two graphs are isomorphic we use the construction from Section 5.1.

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 [26]. This implementation is in C [23]. 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 that can be used to proof theorems in GAP. For this callback functions for entering nodes and (partial) refining were added and calls to these were added to the nauty functions. Where possible these were added in different files or otherwise surrounded by `#ifdef GRAPHISO` to indicate which code is McKay's.

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 included in GAP with the automatically loaded package GAPDoc. The resulting tree is then traveled recursively and transformed into a human readable proof using the functions created for Luks to give structured output. At the moment a lot of information is sent from dreadnaut to GAP in this way. It should be possible to reduce this amount to improve performance.

4.4 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.

Now look at the graphs in Figure 4.2. 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 4.3.

The search tree in Figure 4.4 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

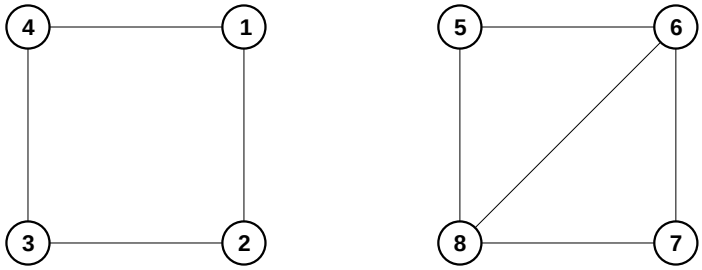

Figure 4.2: Two graphs

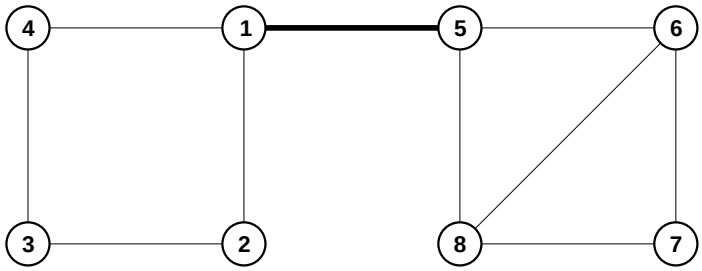

Figure 4.3: The two graphs connected by an edge

the starting partition can be refined in the same way and we will not prove this for each refining step.

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.

Algorithm Output

```
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:
    Proposition: the orbit of x = 1 under A, the group generated by a1 = (2,4) and a2 = (1,3), is X = [ 1, 3 ].
    Proof:
      By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A leaves X
      invariant. It remains to show that the points in X are images of x under elements of A:
```

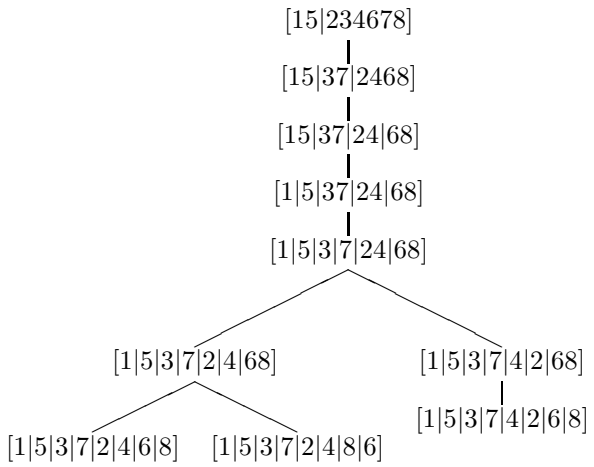

Figure 4.4: The search tree in McKay's algorithm

```
          1 = 1^()
          3 = 1^(1,3) = 1^a2
      QED(orbit of x = 1)
      Proposition: the orbit of x = 2 under A, the group generated by a1 = (2,4) and a2 = (1,3), is X = [ 2, 4 ].
      Proof:
        By the straightforward check that the cycles containing points of X do not contain points not in X, it follows that each generator of A leaves X
        invariant. It remains to show that the points in X are images of x under elements of A:
          2 = 2^()
          4 = 2^(2,4) = 2^a1
      QED(orbit of x = 2)
      It is straightforward to verify that the orbits [ 1, 3 ] and [ 2, 4 ] are disjoint and their union is [ 1, 2, 3, 4 ], so we are done.
    QED(all orbits)
    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.
          QED(the permutations are automorphisms)
          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).
                  Then p^-1p' = (2,4) is an automorphism.
                  Further more it is included in H (it is a generator of H).
                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 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 2
        of H with a new edge.
        The resulting graph F has vertices [ 1 .. 8 ], edges [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8 ], [ 6, 7 ], [ 6, 8 ], [ 7, 8
        ] ] and new coloring [ 1 6 | 2:5 7 8 ].
        We now calculate the automorphism group of F and check whether there exists an automorphism that transforms 1 to 6.
        The automorphism group of the coloured graph G with vertices [ 1 .. 8 ] and edges [ [ 1, 2 ], [ 1, 4 ], [ 1, 6 ], [ 2, 3 ], [ 3, 4 ], [ 5, 6 ], [ 5, 8
        ], [ 6, 7 ], [ 6, 8 ], [ 7, 8 ] ] and colored by the partition [ 1 6 | 2:5 7 8 ]is generated by the permutations[ (5,7), (2,4) ].
        Proof:
          Lemma: The permutations [ (5,7), (2,4) ] are automorphisms.
          Proof:
            This is straightforward to verify.
          QED(the permutations are automorphisms)
          Any automorphism can be written in the form p^-1 p', with p a fixed permutation and p' a variable permutation.
          Let p be [ 6 | 1 | 3 | 2 | 4 | 5 | 7 | 8 ] i.e. (1,6,5,4,2) in cycle notation.
          If [ 1 2 | 3:8 ]^p' = [ 1 6 | 2:5 7 8 ] then [ 1 | 2 | 3 | 4 5 | 6 7 | 8 ]^p' =[ 6 | 1 | 3 | 2 4 | 5 7 | 8 ].
          Proof:
            Lemma (refine part)
              If [ 1 2 | 3:8 ]^p' = [ 1 6 | 2:5 7 8 ] then [ 1 2 | 3 | 4:8 ]^p' = [ 1 6 | 3 | 2 4 5 7 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 6 ].
                [ 3:8 ]^p = [ 2:5 7 8 ].
                The vertex 3 is not connected to any vertices of [ 1 6 ].
                The vertices 2 4 5 7 8 are each connected to 1 vertex of [ 1 6 ].
              QED(p)
              Then for p'
                [ 1 2 ]^p' = [ 1 6 ].
```

```
                [ 3:8 ]^p' = [ 2:5 7 8 ].
                The vertex 3 is not connected to any vertices of [ 1 6 ].
                The vertices 2 4 5 7 8 are each connected to 1 vertex of [ 1 6 ].
              QED(p')
              If p^-1p' is an automorphism then it transfers [ 1 6 ] to [ 1 6 ] and [ 2:5 7 8 ] to [ 2:5 7 8 ] and must therefore transfer [ 3 ] to [ 3 ] and [
              2 4 5 7 8 ] to [ 2 4 5 7 8 ].

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

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

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

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

```
  QED(case distinction)
QED(graphisomorphism)
```

Chapter 5

Proof assistant

5.1 Introduction

We have developed a Proof Assistant for Graph Nonisomorphism. This software package automatically constructs a proof of (non)isomorphism of two given graphs. It is possible to ask for a specific proof by choosing invariants or calling Luks' algorithm or McKay's algorithm.

The software can derive the automorphism group of a single graph by calling the algorithm from Chapter 4. It can further derive a proof of graph nonisomorphism by using the graph automorphism algorithm from Chapter 4 or 3 in the following way.

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, or $G'' = (V'' = V \cup V', E'' = E \cup E' \cup \{v, v'\}, \gamma''(v'') = \{\gamma(v'')$ for $v'' \in V \setminus \{v\}$, $\gamma'(v'')$ for $v'' \in V' \setminus \{v'\}\}$, c, for $v'' \in \{v, v'\})$.

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'.

In this chapter we describe how to work with the proof assistant and give examples. First we describe how to run the program and how to import graphs. We then describe how to determine whether two graphs are isomorphic, or a proof of graph nonisomorphism by invariants, or by using Luks' or McKay's methods. This is followed by predefined examples and information on how to

display vertex invariants. We end this chapter with a discussion of the languages and packages used and a description of the communication between them.

5.2 Starting up and opening graphs

The proof assistant is available both as a stand-alone Java program and as an applet at [31]. At this URL the Java applet starts automatically. It is also possible to download a .zip file with the files needed to run the stand-alone program.

Figure 5.1: Applet certificate dialog

Java applets have fewer permissions than stand-alone programs. To get around this restriction the Java code has been signed. This means that a self-created key-pair has been used to authenticate the code, see [19]. It is also possible to let a trusted third party issue this key-pair, but we have notdone this. When the applet is started up, the dialog as displayed in Figure 5.1 opens. If the user selects "yes" then he gets the full functionality (by giving us the possibility to import files from his local file system, retrieving files from global URLs, and by allowing pasting to a dialog box). If the user selects "no", he can only open relative URLs or type graphs in the dialog box. The ways of entering graphs are described below. The images shown in the remainder of this chapter come from the stand-alone version.

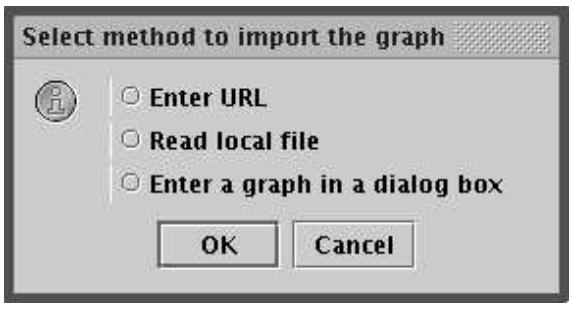

Figure 5.2: Import graph dialog

If a user wants to open a graph he can click one of the "Read left graph" or "Read right graph" buttons. Then a dialog appears (see Figure 5.2) where the user can indicate whether he wants to enter a URL, read a local file or enter a graph in the dialog box.

In the URL dialog, see Figure 5.3, the user can import a graph from a file. In the local file dialog, see Figure 5.4, the user can import a graph from his local file system. The proof assistant

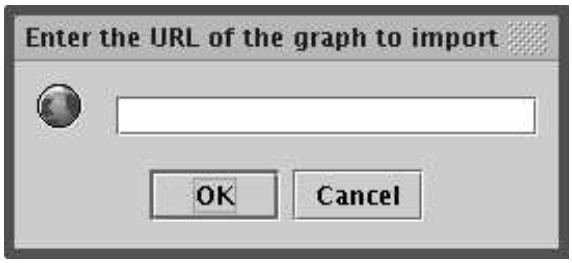

Figure 5.3: Read URL dialog

supports three file formats: an XML-encoded Java graph object with suffix XML, a graph in GXL [18] format, and a graph in a nauty-compatible format. In Section 5.8 it is described how to create a graph in GXL format with GXL Graphpad [15]. The syntax of the nauty-compatible format is described in Section 5.9.

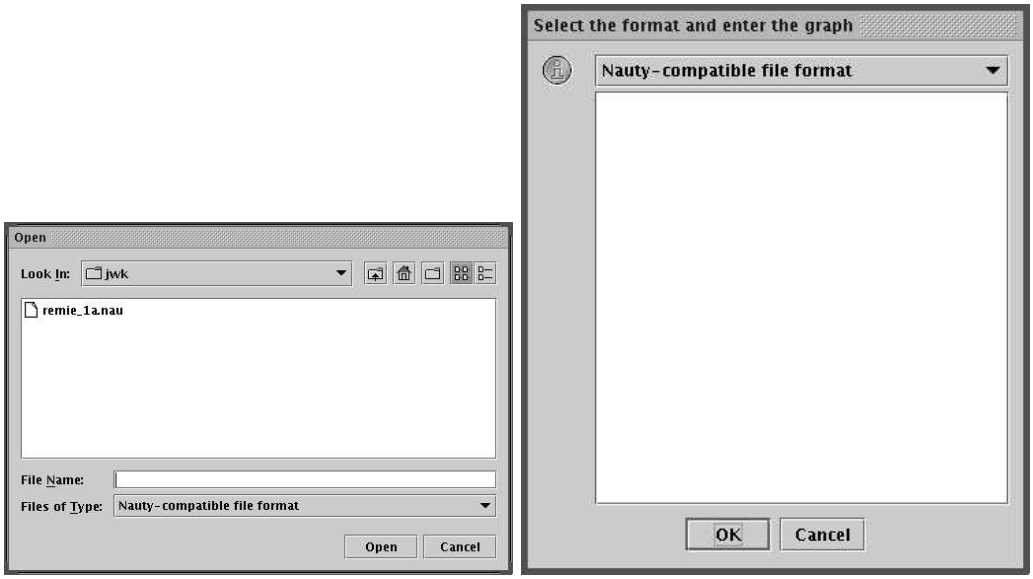

Figure 5.4: Read file dialog and enter file dialog

It is also possible to enter graphs in a dialog box. Unfortunately, if the proof assistant is run in applet mode and the certificate is not accepted, it is not possible to paste text to here. The three file formats described above are all supported, but the most useful is probably the nauty-compatible file format.

5.3 Graph isomorphism for graphs that are not connected

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 possible to reduce graph (non-) isomorphism of unconnected graphs to graph (non-) isomorphism of connected graphs.

First use nauty to get the canonical labelings of the components of both graphs. As long as there is a component in a graphs that is isomorphic to a component in the other graph, prove this with a permutation and remove those components.

If the graphs are isomorphic, all components can be matched to a component in the other

graph. If the graphs are not isomorphic this stops when there are only nonisomorphic components left. It is then sufficient to prove that one of the components of a graph is nonisomorphic to all other components of the other graph.

5.4 Proof assistant example: isomorphic Petersen graphs

In this section we give an example of how the proof assistant works. At the moment there are 8 predefined examples, of which number 7 is the example of two Petersen graphs displayed in different ways, see also Figure 5.5.

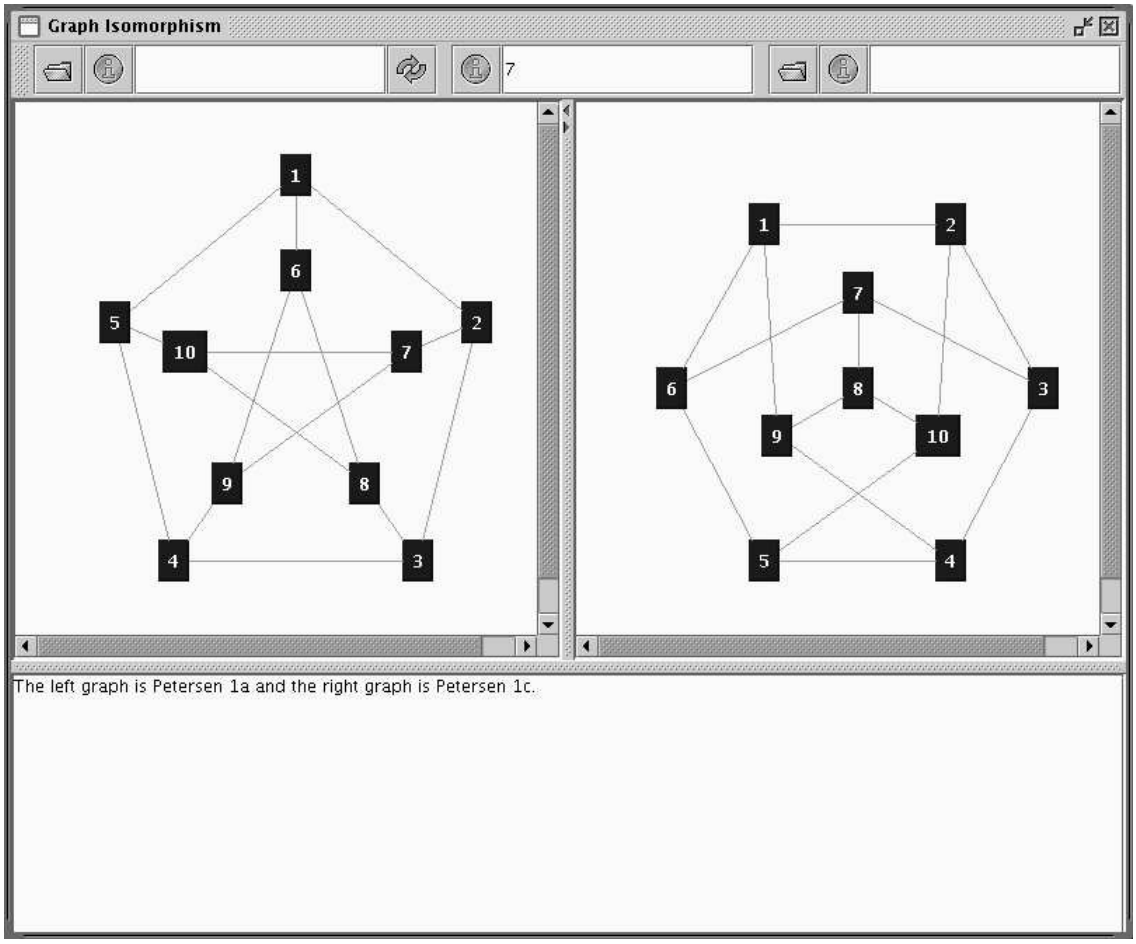

Figure 5.5: Predefined example nr. 7

It is now possible to ask whether the two graphs are isomorphic. This can be done by clicking the "i" button. The program now calls GAP (and possibly nauty) to determine whether the graphs are isomorphic. If they are isomorphic an isomorphism is given. If they are not isomorphic a proof of that fact is given (if known). In the case at hand they are isomorphic and the permutation that transforms the left graph to the right graph is $[1, 2, 3, 4, 9, 6, 10, 7, 5, 8]$ or $(5, 9)(7, 10, 8)$, see also Figure 5.6.

It is now also possible to give an animation that transforms the vertices of the left graph to positions similar to their images in the right graph by pressing the arrows button. The result after the animation is shown in Figure 5.6. The animation is an adaption of the animation used by [32].

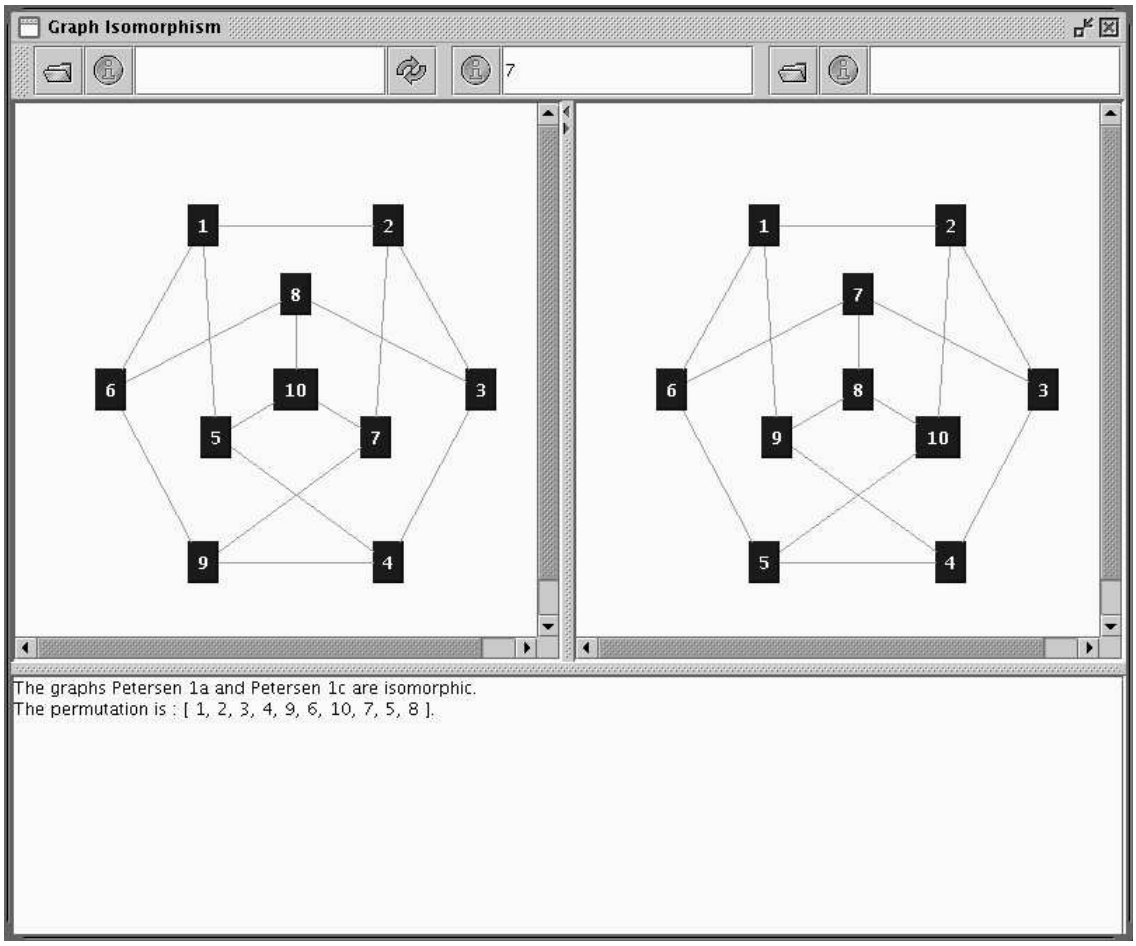

Figure 5.6: After transformation

5.5 Other predefined examples

Now we give some examples of graphs that are not isomorphic. The Java program calls GAP and returns the text given by the GAP code explaining why the two graphs are not isomorphic. In some cases additional information is contained in the proof. This can be information about vertices and or edges with a certain property. The vertices and edges having this property are given a new color. These examples can be found in Figures 5.7 to 5.16

5.6 Vertex invariants

In this section we give an example of how to see which vertices can be separated by vertex invariants. We do this by looking at the graphs with six vertices of Example 3. To compute vertex invariants, right click on any vertex. Then the vertex invariants dialog, displayed in Figure 5.17, pops up. In the dialog select the vertex invariants to calculate. In this example we only select the degree. The result is shown in Figure 5.18.

5.7 Libraries and languages used

The graphical frontend 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 [12] by the corresponding RIACA GAP

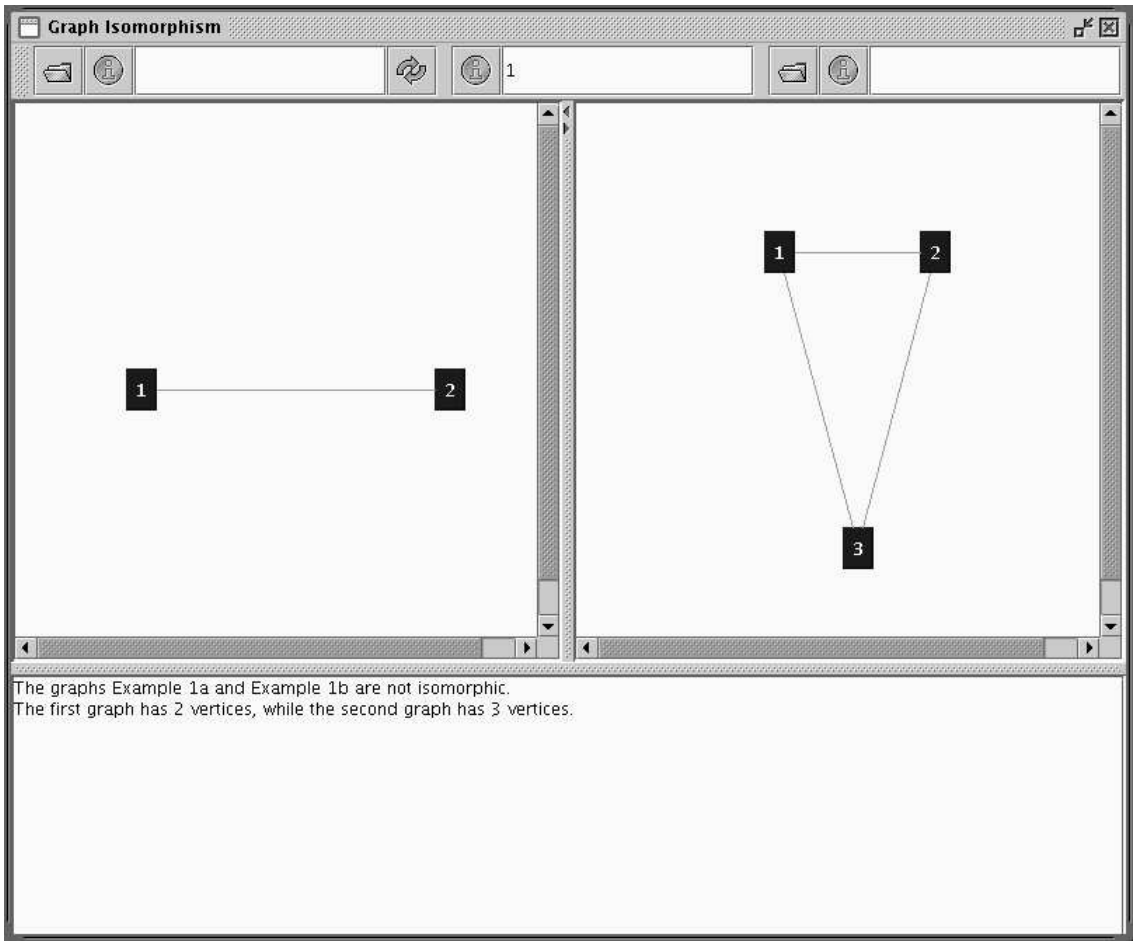

Figure 5.7: Example 1

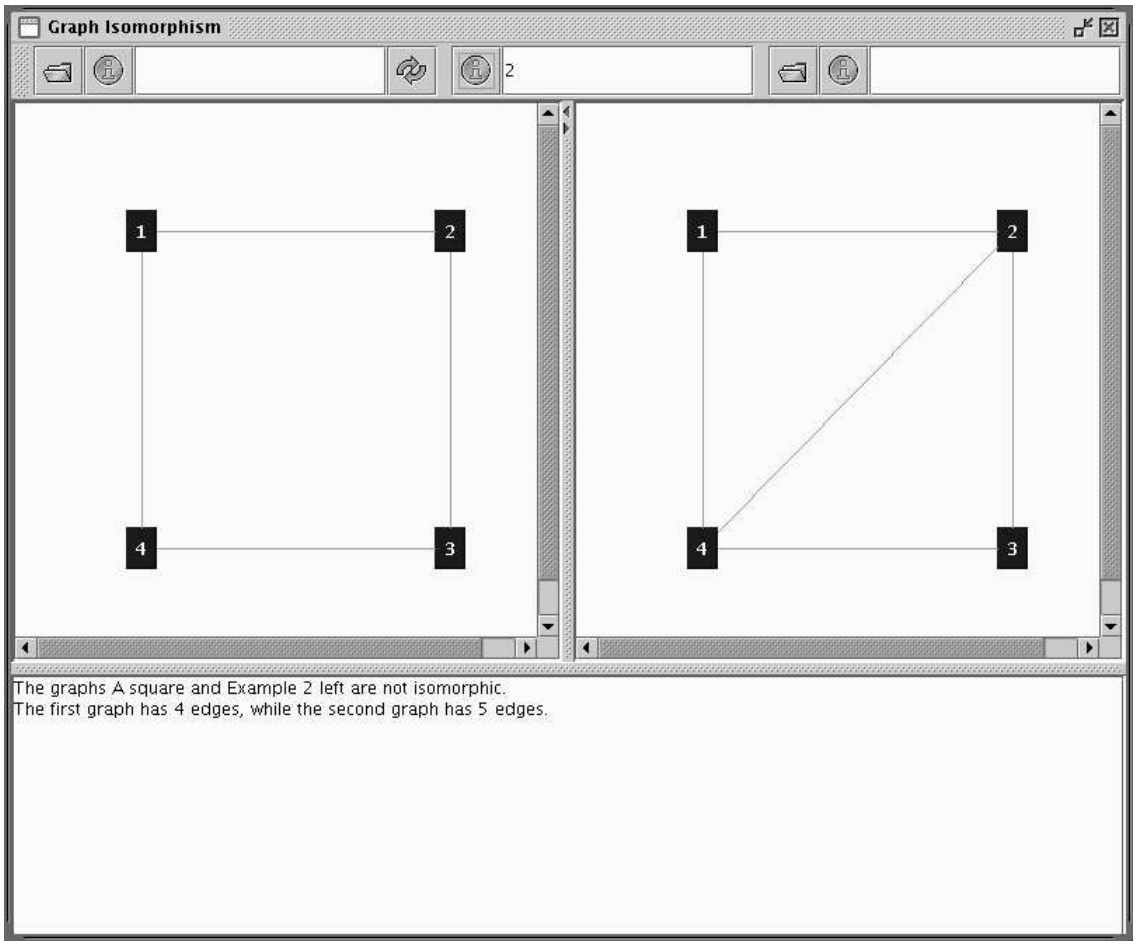

Figure 5.8: Example 2

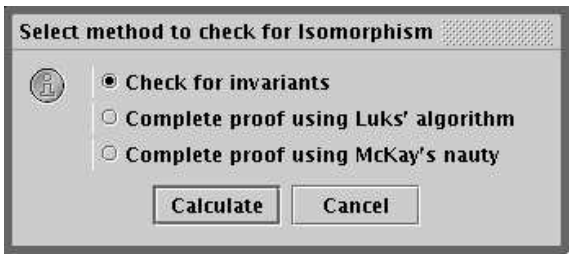

Figure 5.9: Isomorphism dialog

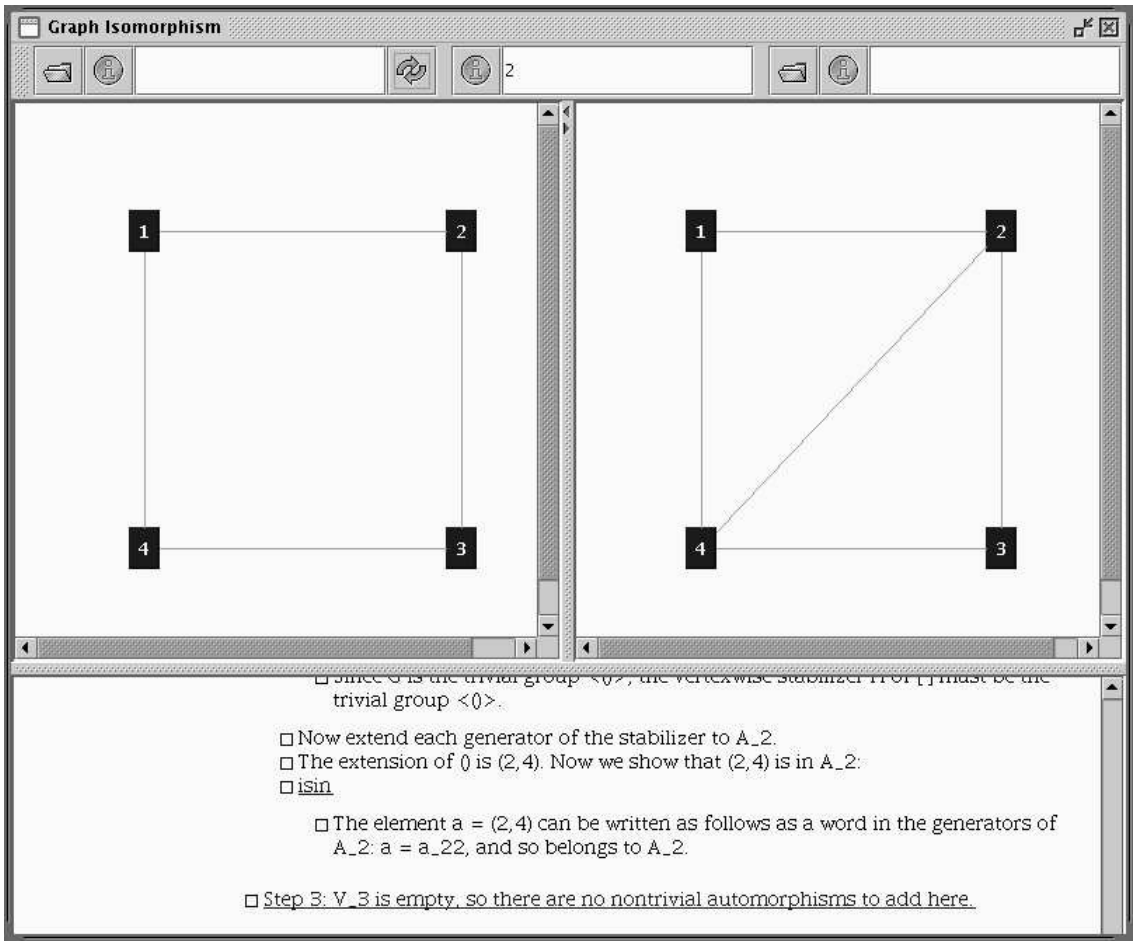

Figure 5.10: Isomorphism proof: Luks

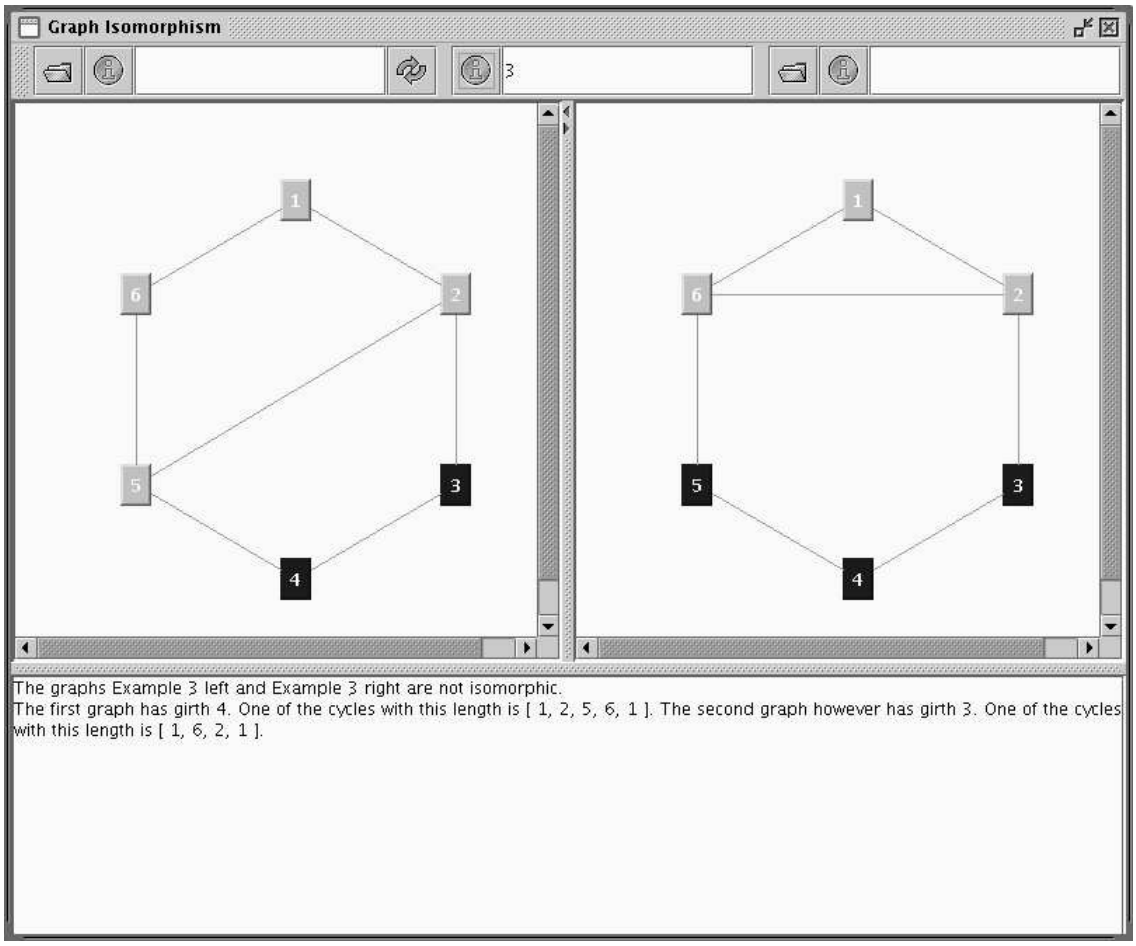

Figure 5.11: Example 3

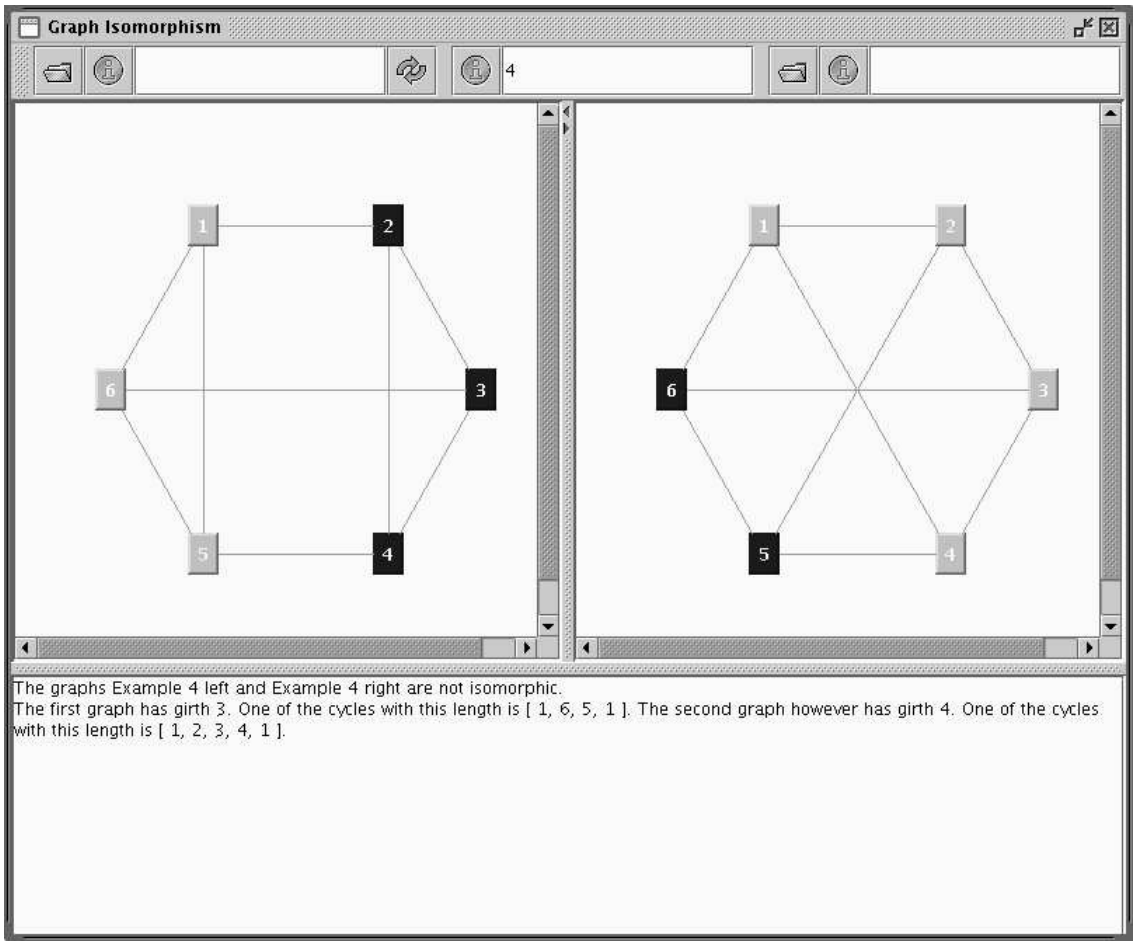

Figure 5.12: Example 4

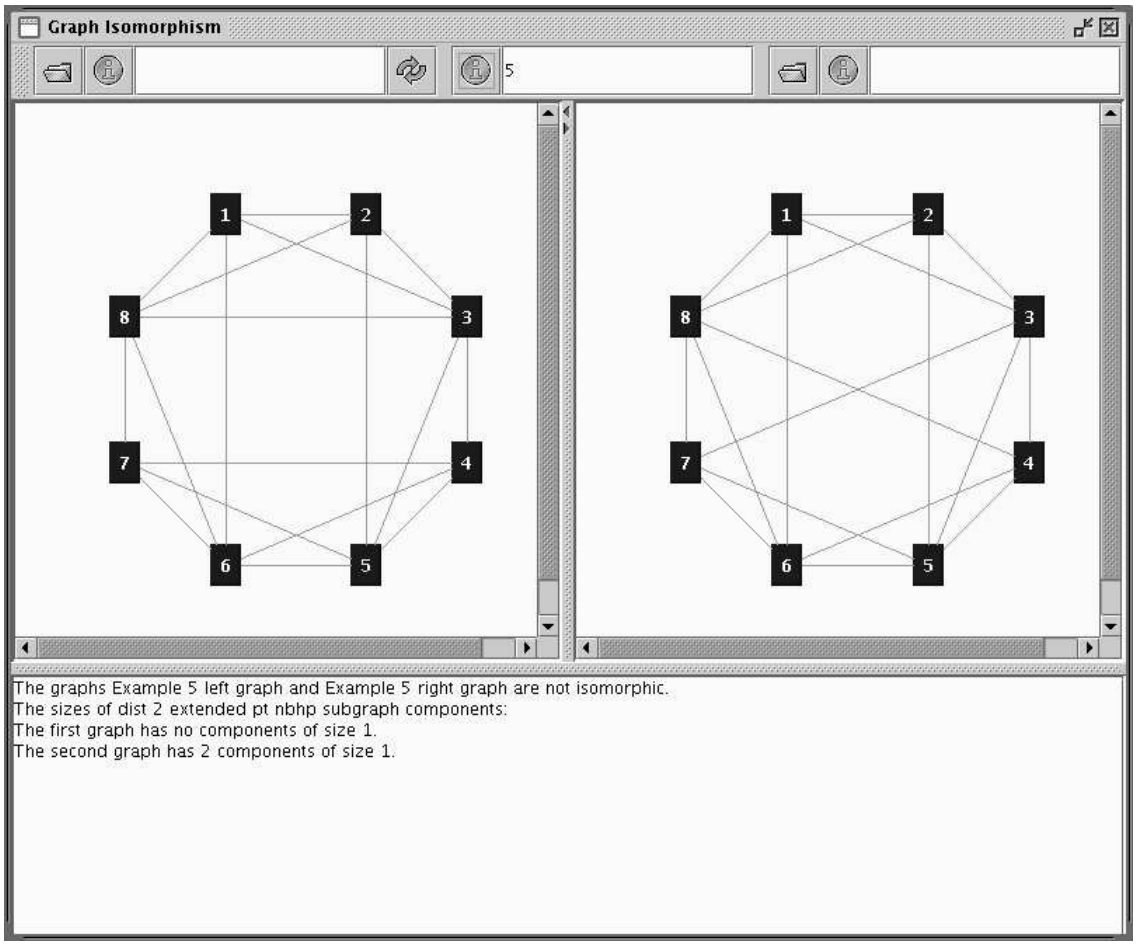

Figure 5.13: Example 5

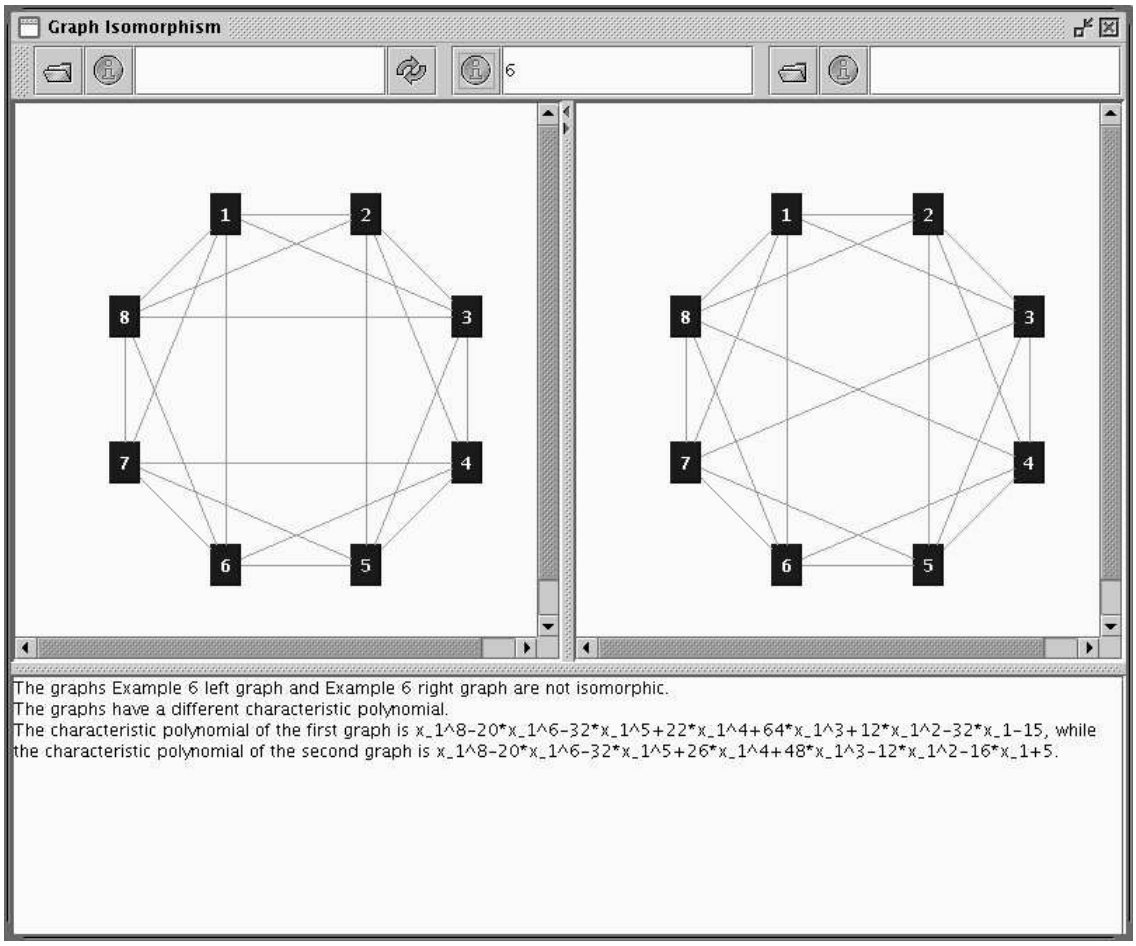

Figure 5.14: Example 6

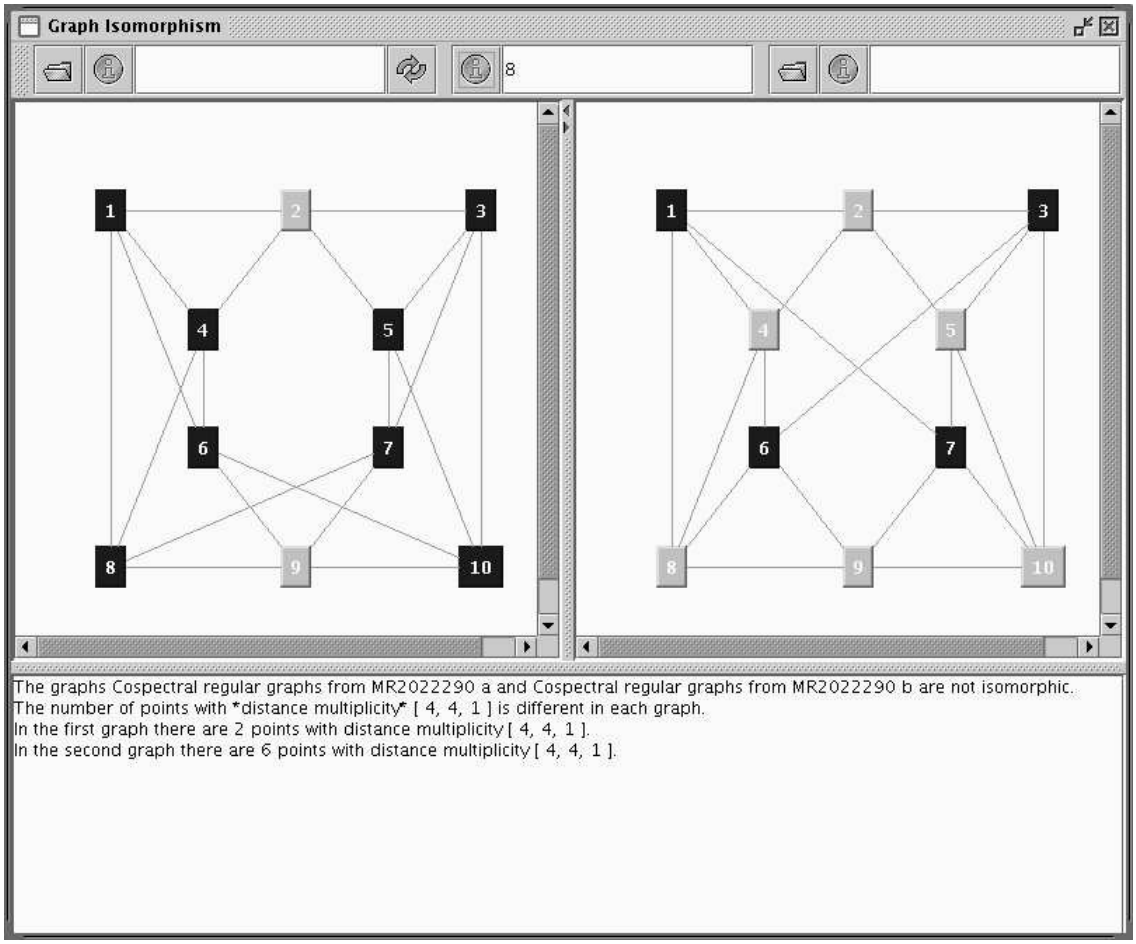

Figure 5.15: Example 8, two graphs from [36]

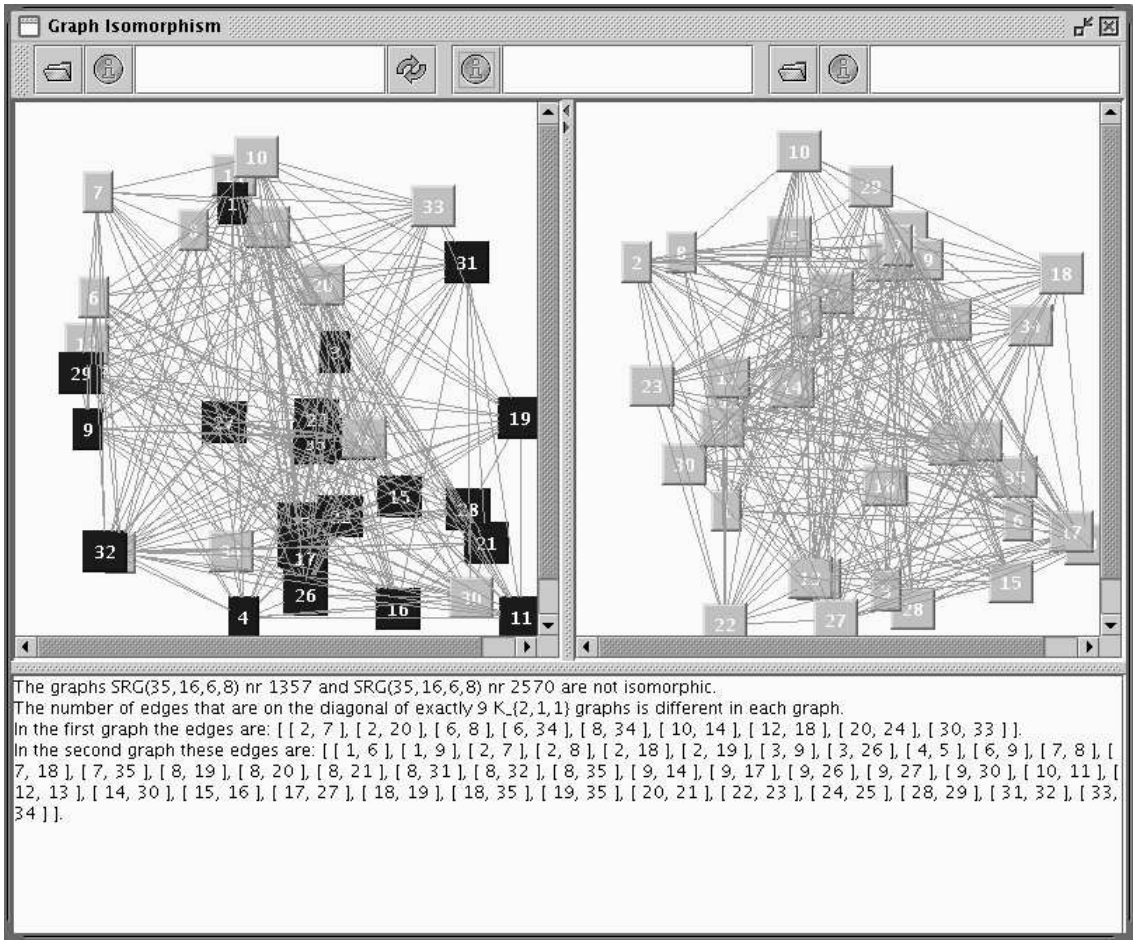

Figure 5.16: Example with strongly regular graphs

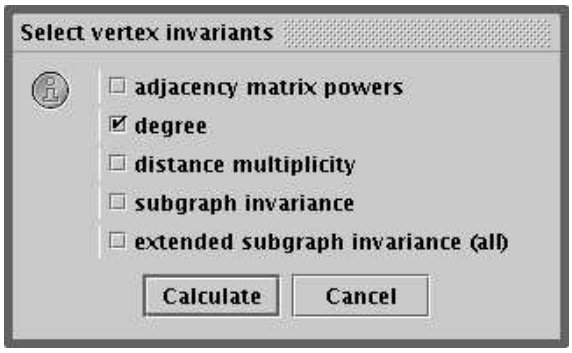

Figure 5.17: Selecting vertex invariants

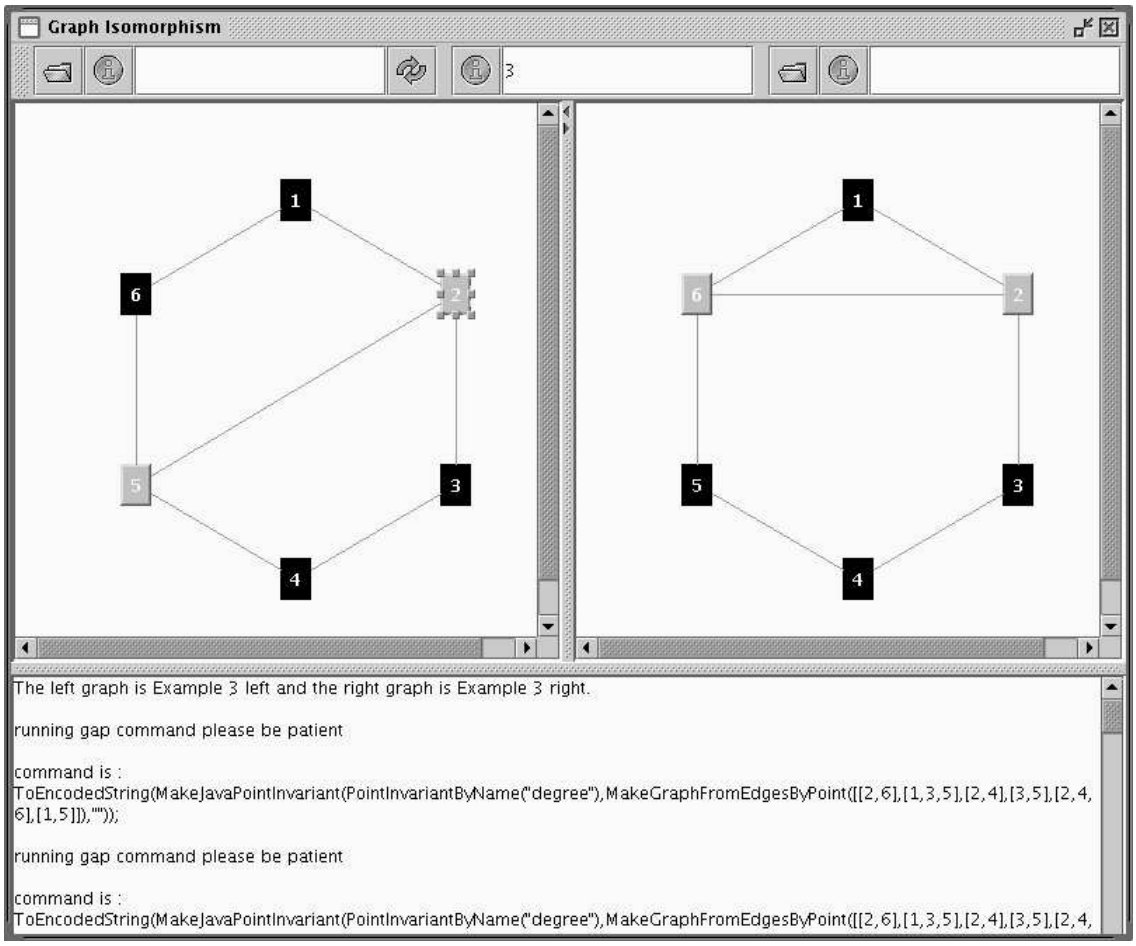

Figure 5.18: Result of vertex invariants

Link [10]. 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 Java program we use the packages listed below. We use the graphs libraries JGraph [20] and JGraphT [21]. Furthermore we use Mapplet [32] for the morphing. For the link with GAP we use the openmath library [30] and GAP phrasebook [11] from RIACA. They depend on the parsing library ANTLR [1]. For support of GXL [18] we use the GXL core package [14]. For the buttons we use the icons from the Java Look and feel Graphics Repository [22].

5.8 Creating graphs: GXL GraphPad

It is possible to use the GXL Graphpad program [15] to construct graphs. If a graph is constructed with the GXL graph model then it is possible to save as a `.gxl` file with an accompanying `.gxllayout` file. These files can be read from the proof assistant. The position of the vertices will be read from the layout file if it exists. Note that the GAP code requires numerical node names $\{1, \ldots, n\}$ and that therefore the nodes in the GXL should be named from 1 to n.

5.9 The nauty-compatible graph format

The number of vertices is set with `n=`$|V|$. The number of the first vertex (in our case one) is set with `$=`n. The neighbors $n_1, \ldots, n_i$ of a vertex v are set with a line `v:` $n_1 \ldots n_i$`;`. In the comments (a comment starts with a `!` at the beginning of the line) information about the name and the layout of the graph can be entered.

```
n=4
$=1
!size(300,300)
g
1:
!pos(50,50)
2 4;
2:
!pos(250,50)
3 4;
3:
!pos(250,250)
4;
4:
!pos(50,250)
;
```

Chapter 6

Conclusion

While a lot of graphs can be distinguished by the invariants listed in 2.6, some graphs can not. A few examples can be found in Section 6.1. It is possible to add more and more invariants, but there will always be graphs that cannot be told apart. Furthermore, some invariants provide proofs that are less easy to understand by a human or harder to calculate. There are some invariants on colored graphs, that are used in nauty that might reduce computation in Chapter 4. It might be useful to see whether the nauty source code can be studied and adapted in such a way that these give a refinement-like proof.

The methods we used to give a general proof are not very fast and give long proofs. It might be possible to make them a bit faster, but the algorithms implemented in the way they currently are are not useful for comparing graphs with a lot of edges and much more than 10 vertices.

The hierarchical structure of the proofs can be used to display the proof with structure: users can click on a statement if they want to see a proof and otherwise don't have to read the proof. But the fact that such a feature is useful indicates that

The files and information on the structure and how to install them can be found online at [31]. The files have also been added to the RIACA software repository.

6.1 Graphs that can not be distinguished by invariants

Some graphs still can not be handled by the proof assistant. An example is a set of four graphs on 24 vertices constructed by Brouwer with his classification of $(0, 2)$-graphs [2]. These graphs can be found with the gap package (`graphs/brouwer1.g`, `graphs/brouwer2.g`, `graphs/brouwer3.g` and `graphs/brouwer4.g`). These graphs can be distinguished by the number of occurences of the subgraph from Figure 6.1.

Some other set of graphs that cannot be distinguished by our approach are the strongly regular graphs of type $(35, 16, 6, 8)$ [29]. Of the 3853 graphs, only 3486 different values of the list of invariants can be distinguished. An example of two graphs that cannot be distinguished are the graphs 331 and 2101. The example `examples/groups.g` can be used to obtain these numbers and find all pairs of graphs that take the same value for all of our invariants.

6.2 Things learned from implementing Luks' algorithm

Luks' algorithm was implemented first, because it was an logical extension of the permutation algorithms already available. During this implementation functions were implemented to format and structure the proof. Some of the algorithms are run twice, first to generate a result and then to proof the result. It is sometimes possible to only run the algorithm once, and use a temporary string variable to a store part of the proof.

The formatting functions were also used when implementing McKay's algorithm. The algorithm is run first (in dreadnaut) and therefore all information is available. Like with Luks'

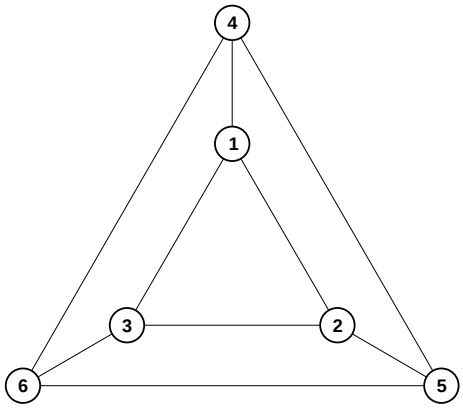

Figure 6.1: The number of these prism-like subgraphs can be used as an invariant

algorithm we only prove the things necessary to gain a proof. Some proofs like proving that a partition can not be refined further are not necessary. Last but not least, the algorithm to prove graph isomorphism with graph automorphism came from the implementation of Luks' algorithm.

6.3 Suggestions

There are several things that might be interesting, but that we couldn't do because due to lack of time. It might be interesting to transform the algorithms so that they can generate computer readable proofs as in [4]. We described how the algorithms of Luks and McKay could be used with colored graphs, but did not implement this. This should not be hard to do. The algorithm by McKay also works for directed graphs and therefore it should not be hard generate a proof of nonisomorphism of directed graphs using a modified version of nauty. In Section 5.3 we describe a method to determine (non-) isomorphism of graphs that are not connected. It should be straightforward to implement an algorithm that gives a proof for this. The invariants included in nauty are currently not used. Most of these can probably be transformed so that a proof can be given, which can reduce the search tree and may reduce the length of the proof. At the moment a lot of data is sent from nauty to gap. This seems to be a bottleneck. It is possible to reduce the amount of data and replace it by checks and minor calculations. This probably will allow processing larger graphs.

Bibliography

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

[2] Andries E. Brouwer. Classification of small (0,2)-graphs. Preprint, oktober 2005.

[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`.

[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.

[5] Arjeh M. Cohen, Hans Cuypers, and Hans Sterk, editors. *Some tapas of computer algebra*, volume 4 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, 1999.

[6] P. M. Cohn. *Algebra, Vol. 1.* John Wiley & Sons, London-New York-Sydney, 1974.

[7] 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.

[8] Dragoš M. Cvetković, Michael Doob, and Horst Sachs. *Spectra of graphs*. Johann Ambrosius Barth, Heidelberg, third edition, 1995. Theory and applications.

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

[10] RIACA GAP phrasebook, 2004. Available from World Wide Web: `http://www.mathdox.org/projects/openmath/lib`.

[11] RIACA GAP phrasebook, 2004. Available from World Wide Web: `http://www.mathdox.org/projects/openmath/lib`.

[12] RIACA GAP phrasebook, 2004. Available from World Wide Web: `http://www.mathdox.org/projects/openmath/lib`.

[13] Jonathan L. Gross and Jay Yellen, editors. *Handbook of graph theory*. Discrete Mathematics and its Applications (Boca Raton). CRC Press, Boca Raton, FL, 2004.

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

[15] *GXL graphpad 0.81*, 2003. Available from World Wide Web: `http://gxl.sourceforge.net/index.html`.

[16] Christian Heilmann. *Collapsible Explorer navigation with pureDOMexplorer (pde)*. Available from World Wide Web: `http://onlinetools.org/tools/puredom/index.html`.

[17] Christoph M. Hoffmann. *Group-theoretic algorithms and graph isomorphism*, volume 136 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1982.

[18] 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/`.

[19] Signing jar files, 2005. Available from World Wide Web: `http://java.sun.com/docs/books/tutorial/jar/sign/signing.html`.

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

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

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

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

[24] Donald L. Kreher and Douglas R. Stinson. *Combinatorial Algorithms: Generation, Enumeration, and Search*. CRC Press, Boca Raton-London-New York-Washington, DC, 1999.

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

[26] 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.

[27] 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.

[28] 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.

[29] Brendan D. McKay. Combinatorial data, 2006. Available from World Wide Web: `http://cs.anu.edu.au/~bdm/data/graphs.html`.

[30] RIACA openmath library, 2004. Available from World Wide Web: `http://www.mathdox.org/projects/openmath/lib`.

[31] Proof assistant for graph non-isomorphism, 2006. Available from World Wide Web: `http://www.stack.nl/~jwk/graphs/`.

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

[33] Vincent Remie. Graph isomorphism problem. Bachelors project, Eindhoven University of Technology, Department of Industrial Applied Mathematics, September 2003.

[34] Ákos Seress. *Permutation group algorithms*, volume 152 of *Cambridge Tracts in Mathematics*. Cambridge University Press, Cambridge, 2003.

[35] 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/`.

[36] Edwin R. van Dam and Willem H. Haemers. Which graphs are determined by their spectrum? *Linear Algebra Appl.*, 373:241–272, 2003. Special issue on the Combinatorial Matrix Theory Conference (Pohang, 2002).

Index