

A Group-Theoretic Isomorphism Program
for 3-Regular Graphs

by

H. A. Cameron

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science

© H. A. Cameron 1985

A GROUP-THEORETIC ISOMORPHISM PROGRAM FOR 3-REGULAR GRAPHS

BY

HELEN A. CAMERON

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

© 1985

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

A Group-Theoretic Isomorphism Program for 3-Regular Graphs

This thesis is dedicated to Glen Ditchfield and to Bill Kocay. Without these two people, this thesis would neither have been thought of nor completed!

Abstract

This thesis describes a group-theoretic algorithm for finding the automorphism group of a 3-regular graph, and the actual computer implementation of the algorithm in Pascal.

Chapter 1 contains the basic terminology and ideas from group theory and graph theory that are necessary to develop the algorithm.

Chapter 2 describes Schreier's algorithm, which takes a set of generators for a group G and expands G as a tower of point stabilizers. A non-redundant set of generators for each group and a set of transversal elements for the cosets of the next subgroup down in the tower are found.

Chapter 3 describes how finding the automorphism group acting on a graph X reduces to finding a subgroup $\text{Aut}_e(X)$ of the automorphism group fixing an edge e . To find $\text{Aut}_e(X)$, we partition the vertices of X into $V_0 + V_1 + \dots + V_h$, where V_i contains all vertices at distance i from an imaginary point v_0 on the chosen edge e . Subgraphs $X_0, X_1, X_2, \dots, X_{h+1}$ are defined, where X_k has vertex set $V_0 + V_1 + \dots + V_k$ and edges $X[V_0 + V_1 + \dots + V_k]$ except for the edges connecting vertices in V_k . Starting with group $G_0 = \{e\}$, we find $G_k = \text{Aut}_e(X_k)$ for each subgraph X_k , until $G_{h+1} = \text{Aut}_e(x)$ is found.

Chapter 4 shows how to find a set-wise stabilizer of set M in a p -group G which is known to set-wise stabilize a set X containing the set M . If G is intransitive on the set X , we use the orbits of G in X to reduce the problem to a set of smaller problems. If G is transitive on the set X , we use the fact that p -groups have systems of imprimitivity to reduce the problem to a set of smaller problems. The basic properties of p -groups are also included.

Chapter 5 analyzes the complexity of the algorithm to find $\text{Aut}_e(X)$. The total complexity is $O(n^7 \log n)$, although this is a very worst-case estimate.

Chapter 6 describes the implementation in Pascal. It also contains a listing of the main

A Group-Theoretic Isomorphism Program for 3-Regular Graphs

routines of the program and a sample run.

Table of Contents

1. Introduction.....	1
2. Schreier's Algorithm.....	8
3. Isomorphism Testing in 3-Regular Graphs.....	22
4. Set-Wise Stabilisers.....	34
5. The Complexity of the Algorithm.....	53
6. The Program.....	60
7. Bibliography.....	67

1. Introduction

The purpose of this thesis is to study the graph isomorphism problem. The problem consists of determining whether two graphs are essentially the same graph; only the names of the vertices and edges may differ. In other words, the problem is to determine if there is an isomorphism mapping the vertices of one graph to the vertices of the other which preserves the edge set.

In particular, this thesis examines the graph isomorphism problem from a group-theoretic point of view. From this point of view, the graph isomorphism problem becomes the problem of finding the automorphism group acting on a graph. If we restrict ourselves to 3-regular graphs, we find that the groups we work with all have a special property which simplifies the problem: the groups are 2-groups. This thesis follows the approach taken by Furst, Hopcroft and Luks[3] to find the subgroup of the automorphism group which fixes an edge of a 3-regular graph, X . The subgroup is denoted $\text{Aut}_e(X)$. This gives us a polynomial algorithm for the graph isomorphism problem restricted to 3-regular graphs.

Although this method allows us to study the graph isomorphism problem restricted to 3-regular graphs, it does have some drawbacks. One of the drawbacks of this approach can be seen by attempting to extend the algorithm to handle graphs of higher degree. For a graph X of higher degree, $\text{Aut}_e(X)$ is no longer necessarily a p -group. Thus, we would be forced to work with the Sylow p -subgroups of $\text{Aut}_e(X)$, and the algorithm would quickly become very complicated.

Also, in our method, we read in two graphs and determine if they are isomorphic. If we had n graphs to test for isomorphism, we would have to call our algorithm once for each pair of graphs. For large values of n , this is too expensive.

Another approach would be to read in one graph X and produce a certificate, $\text{cert}(X)$,

for X . $\text{Cert}(X)$ is an object such that graph X is isomorphic to graph Y if and only if $\text{cert}(X)$ is equal to $\text{cert}(Y)$. To test n graphs for isomorphism we would only need to find their certificates and sort them. However, the polynomial algorithms known for this approach are limited to a very few classes of graphs.

1.1 Background Material

Since the topic of this thesis is a group-theoretic graph isomorphism program, some knowledge of graph theory and group theory is required. This chapter provides some background material for the following chapters. The coverage of graph theory is slight. The graph-theoretic notation used is generally that of Bondy and Murty [2], and that text may be referred to by the reader for more information. The group-theoretic background provided is more complete and is covered in more detail in Hall [4].

A graph G consists of a non-empty set $V(G)$ of vertices, another disjoint set $E(G)$ of edges, and an incidence function Ψ_G that associates every edge of G with an unordered pair of vertices of G .

A walk W in a graph G is a sequence of the form $v_0 e_1 v_1 e_2 v_2 \dots e_n v_n$ where $v_i \in V(G)$ for $i = 0, 1, 2, \dots, n$, and v_{j-1} and v_j are the vertices associated with edge e_j .

If $e_1, e_2, \dots, e_n$ of walk W are distinct, then W is a trail.

If $v_0, v_1, v_2, \dots, v_n$ of walk W are also distinct, then W is a path.

If $v_0 = v_n$ and $v_0, v_1, v_2, \dots, v_{n-1}$ are distinct, then W is a cycle.

A tree is a connected graph containing no cycles.

A spanning tree T of graph $G=(V(G),E(G))$ is a subgraph of G which is a tree and such that $V(T) = V(G)$.

Lemma. In a tree, any two vertices are connected by a unique path. [2]

Let H be a graph with spanning tree T . Each $e \in E(H) - E(T)$ is a chord of H with

respect to T . For each chord e of H with respect to T , $T+e$ contains a unique cycle C_e , the fundamental cycle of e with respect to T .

Let G be a group of permutations which permute the points in set X . Permutations are multiplied from left to right, so that if $a, b, c \in G$ and $x \in X$, then the action of the product abc on x is $(x)abc = ((x)a)b)c$.

Let $k \in X$. The point-wise stabilizer of k in G , $\text{Stab}(k)$, is the subgroup of G which fixes k : i.e. $\text{Stab}(k) = \{ g \in G \mid (k)g = k \}$.

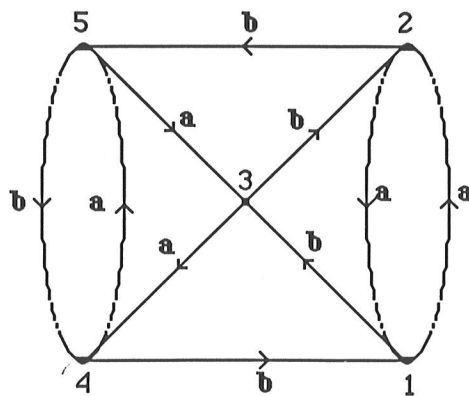
Let M be a subset of X . Then the set-wise stabilizer of M in G , $\text{Stab}(M) = \{ g \in G \mid \forall k \in M: (k)g \in M \}$.

For permutations $a_1, a_2, \dots, a_n$ permuting a set X , the permutation diagram is a directed graph. The vertices of the graph are the elements of the set X . For each of the permutations a_i , there is a directed edge from each $k \in X$ joined to $(k)a_i$, labelled a_i . For example:

$$X = \{1, 2, 3, 4, 5\}$$

$$a = (1\ 2)(3\ 4\ 5)$$

$$b = (1\ 3\ 2\ 5\ 4)$$



Let K be a subgroup of G , written $K \leq G$. Choose a complete set of coset representatives, $\{ a_1, a_2, \dots, a_n \}$, for the cosets of K in G , i.e. a transversal for K in G .

Consider coset Ka_jg , where $g \in G$. This is coset Ka_j for some j . g defines a permutation, g^* , of the cosets of K in G . The right representation of G by cosets of K is the permutation group $G \bmod K = \{ g^* \mid g \in G \}$.

The right regular representation of G is $G \bmod \{e\}$. Since $\{e\}g = \{e\}h$ if and only if $g = h$, $G \bmod \{e\}$ is a group of degree $|G|$ and $|G \bmod \{e\}| = |G|$.

Let G be a group generated by the permutations $b_1, b_2, \dots, b_s$, written $G = \langle b_1, b_2, \dots, b_s \rangle$. The Schreier coset diagram for group G with respect to $K \leq G$ is the permutation diagram of $b_1^*, b_2^*, \dots, b_s^*$ in the right representation of G by cosets of K . This is also denoted $G \bmod K$.

Let $G = \langle b_1, b_2, \dots, b_s \rangle$. The connected components of the permutation diagram of the permutations $b_1, b_2, \dots, b_s$ are the orbits of G .

G is transitive on X if G has only one orbit. Otherwise, G is intransitive on X .

Let $G = \langle b_1, b_2, \dots, b_s \rangle$ and $K \leq G$. A word in the generators of G is a sequence of the symbols $b_1, b_2, \dots, b_s, b_1^{-1}, b_2^{-1}, \dots, b_s^{-1}$, e.g. $b_1 b_2 b_2 b_1^{-1} b_2^{-1}$. A word can be thought of as defining a walk in the Schreier coset diagram, $G \bmod K$. Choose a coset, Ka_i , as a starting point. Follow the edges in the Schreier coset diagram as indicated by the symbols in the word, in the order that they appear in the word. For example, if the word $w = b_1 b_2 b_1^{-1}$, then we could think of w as the walk in $G \bmod K$ starting at Ka_i , following the edge labelled b_1^* to $(Ka_i)b_1^* = Ka_i b_1$, then following the edge labelled b_2^* to $(Ka_i b_1)b_2^* = Ka_i b_1 b_2$, and finally, proceeding backwards along the edge labelled b_1^* , which ends at $Ka_i b_1 b_2$, to end the walk at $Ka_i b_1 b_2 b_1^{-1} = Ka_j$ for some j . Similarly, a walk P from Ka_i to Ka_j in $G \bmod K$ defines a word, denoted $w(P)$, in the generators of G . Note that $w(P^{-1}) = (w(P))^{-1}$, where P^{-1} indicates the opposite traversal of P (from Ka_j to Ka_i).

The conjugate of subset H of group G by $g \in G$ is $H^g = g^{-1}Hg$.

A subgroup K of group G for which $K^a = K$ for all $a \in G$ is a normal subgroup of G ,

written $K \diamond G$.

Let $K \diamond G$. Let $\{Ka_1, Ka_2, \dots, Ka_r\}$ be a complete set of cosets of K in G . The factor group G/K consists of the cosets of K together with the multiplication rule $Ka_i \cdot Ka_j = Ka_ia_j$.

A homomorphism is a mapping θ from a group G to a group H such that, $\forall g_1, g_2 \in G$, $(g_1 \cdot g_2)\theta = (g_1)\theta \cdot (g_2)\theta$. The kernel of θ , $\text{Ker}(\theta)$, is the set of all elements of G which are mapped by θ onto the identity of H . The image of θ , $\text{Im}(\theta)$, is the set of all elements of H which are the images under θ of at least one element of G .

An isomorphism is a homomorphism $\theta : G \rightarrow H$ that is both 1:1 and onto. In this case, G is said to be isomorphic to H , written $G \cong H$.

Consider the factor group $G/\{e\} = \{ \{e\}g \mid g \in G \}$. Let $\pi : g \in G \rightarrow \{e\}g \in G/\{e\}$. Clearly π is an isomorphism. Therefore, $G \cong G/\{e\}$.

Theorem. Let G and H be groups and let $\theta : G \rightarrow H$ be a homomorphism. Then $\text{Im}(\theta) \leq H$, $\text{Ker}(\theta) \diamond G$, and $G/\text{Ker}(\theta) \cong \text{Im}(\theta)$.

Lemma. Let G and H be groups permuting a set X . Let $K \leq H$. Let $\pi : G \rightarrow H$ be an onto homomorphism. Then G contains a subgroup M such that $M/\text{Ker}(\pi) \cong K$.

Proof. We have $G/\text{Ker}(\pi) \cong \text{Im}(\pi) = H$ since π is an onto homomorphism.

Let Ψ be an isomorphism from H to $G/\text{Ker}(\pi)$.

The set of all cosets of $\text{Ker}(\pi)$ in $G/\text{Ker}(\pi)$ which are images under Ψ of elements in K form a subgroup of $G/\text{Ker}(\pi)$ isomorphic to K . The union of these cosets form a subgroup M of G such that $M/\text{Ker}(\pi) \cong K$. •

Let G be a group acting on a set X . Then the set of orbits produced by the action of G on X is denoted $\text{Orbits}(G)$.

Lemma. Let $K \leq G$, $a \in G$. Then $K^a \leq G$, and the orbits of K^a are the translates of the orbits of K , i.e. $\text{Orbits}(K^a) = (\text{Orbits}(K))a$.

Proof.

Let B be an orbit of K . Then $(B)a$ is an orbit of K^a ; for $(B)a(a^{-1}Ka) = (B)Ka = (B)a$.

Let D be an orbit of K^a . Then $(D)a^{-1}$ is an orbit of K ; for $(D)a^{-1}K = (D)a^{-1}Kaa^{-1} = ((D)a^{-1}Ka)a^{-1} = (D)a^{-1}$.

$\therefore \text{Orbits}(K^a) = \text{Orbits}(K)a$. •

Note that if $K^a = K$ then $\text{Orbits}(K) = \text{Orbits}(K^a) = (\text{Orbits}(K))a$, i.e. if $K^a = K$ then a permutes the orbits of K .

Let $K^a = K$ and $g \in Ka$. Then $g = ka$ where $k \in K$. $Kg = g^{-1}Kg = (ka)^{-1}K(ka) = a^{-1}k^{-1}Kka = a^{-1}Ka = K^a = K$. Therefore, every element in coset Ka permutes the orbits of K .

If $\theta : G \rightarrow H$ is any homomorphism, then $\text{Im}(\theta)$ is said to be a representation of G . A representation is faithful if the homomorphism is 1:1, i.e. $G \cong \text{Im}(\theta)$.

Let $K \leq G$ and let $\theta : G \rightarrow G \text{ mod } K$ be the homomorphism mapping $g \in G$ to $g^* \in G \text{ mod } K$.

$$\text{Im}(\theta) = \{g^* \in G \text{ mod } K \mid g \in G\} = G \text{ mod } K.$$

$$\text{Ker}(\theta) = \{g \in G \mid \forall Ka_i : (Ka_i)g^* = Ka_i g = Ka_i\}.$$

Let H be the largest subgroup of K which is normal in G , i.e. $H \leq K \leq G$ and $H \triangleleft G$.

Lemma. $G \text{ mod } K \cong G/H$.

Proof. We know that $G/\text{Ker}(\theta) \cong \text{Im}(\theta) = G \text{ mod } K$, and $\text{Ker}(\theta) \triangleleft G$.

Claim: $\text{Ker}(\theta) \leq K$.

Proof: Let $b \in \text{Ker}(\theta)$. Then, for any coset Ka_i of K in G , $(Ka_i)b^* = Ka_i b = Ka_i$. In particular, $(K)b^* = Kb = K$. But $e \in K$, so $eb = b \in K$. •

Claim: $H \leq \text{Ker}(\theta)$.

Proof: Let $h \in H$ and let Ka_i be a coset of K in G . $(Ka_i)h^* = Ka_ih$. But $H \triangleleft G$ and $a_i \in G$, so $a_iH = Ha_i$. Hence $a_ih = h'a_i$, where $h' \in H$. $(Ka_i)h^* = Kh'a_i$. $H \leq K$ so $Kh' = K$.
 $\therefore (Ka_i)h^* = Ka_i$ and $h \in \text{Ker}(\theta)$. •

H is the largest subgroup of K such that $H \triangleleft G$. $\therefore H = \text{Ker}(\theta)$, $G/\text{Ker}(\theta) = G/H$ and $G/H \cong G \text{ mod } K$. •

Lemma. If $K \triangleleft G$ then $|G \text{ mod } K| = [G:K]$.

Proof. Let $K \triangleleft G$. Let $g^* \in G \text{ mod } K$. g^* permutes the cosets of K in G , $Ka_1 + Ka_2 + \dots + (Ka_i)g^* = Ka_i g$.

$G/K = \{Ka_i\}$ together with the rule $Ka_i \cdot Ka_j = Ka_i a_j$. What is the identity in G/K ? It is the coset Ka_i such that $Ka_i \cdot Ka_j = Ka_j \cdot Ka_i = Ka_i$ for every Ka_i in G/K . We know that $Ke \cdot Ka_i = Ka_i$ and $Ka_i \cdot Ke = Ka_i$. Thus the identity in G/K is $Ke = K$.

Consider $G/K \text{ mod } \{K\}$, the right regular representation of G/K . $Ka_i^* \in G/K \text{ mod } \{K\}$ permutes the cosets of $\{K\}$ in G/K . $(\{K\}Ka_j)Ka_i^* = \{K\}Ka_j \cdot Ka_i = \{K\}Ka_j a_i = \{Ka_j a_i\}$.

Thus $g^* \in G \text{ mod } K$ maps Ka_i to $Ka_i g$, while $Kg^* \in G/K \text{ mod } \{K\}$ maps $\{K\}Ka_i = \{Ka_i\}$ to $\{Ka_i g\}$.

Therefore, $|G \text{ mod } K| = |G/K \text{ mod } \{K\}| = |G/K| = [G:K]$. •

Lemma. Let $K \triangleleft G$, and let $K \leq H \leq G$. Then $K \triangleleft H$ and $H/K \leq G/K$.

Proof. Let $h \in H$. Since $H \leq G$, $h \in G$. Therefore, $K^h = K$. Hence, $K \triangleleft H$.

Consider Ka_i , where $a_i \in H$. This is a coset of K in H . Since $H \leq G$, $a_i \in G$ and coset Ka_i is a coset of K in G . Thus $H/K \leq G/K$. •

2. SCHREIER'S ALGORITHM

2.1 The Algorithm

Schreier's algorithm, also known as Sims's algorithm [7], is an important part of the isomorphism program. By removing redundant generators from a set of generators for a group, it is used to reduce the number of generators needed to describe a group. It is also used to stabilize sets containing only one element since the complexity of Schreier's algorithm is smaller than that of the set-wise stabilizing routines. The description of Schreier's algorithm here is similar to [6].

At the heart of the algorithm is the idea of a point stabilizer. Let H be a group whose elements permute the set of points, X . Let $k \in X$. The orbit of k in H is used to represent the cosets of $\text{Stab}(k)$ in H . A coset, $\text{Stab}(k)h$, where $h \in H$, is represented by $(k)h$, a point in the orbit of k in H . If $g \in \text{Stab}(k)h$, then it can be written in the form bh , where $b \in \text{Stab}(k)$, and $(k)bh = (k)h$. So every permutation in coset $\text{Stab}(k)h$ maps point k to $(k)h$. If $g \in H$ maps point k to point $(k)h$, then $(k)gh^{-1} = (k)hh^{-1} = k$. Therefore, $gh^{-1} \in \text{Stab}(k)$ and $g \in \text{Stab}(k)h$. Thus, all permutations mapping point k to point $(k)h$ are in coset $\text{Stab}(k)h$.

Schreier's algorithm represents a group as a tower of point stabilizers. The top group in the representation of H is the group H , itself. The next group down in the tower is the stabilizer of a point $k_1 \in X$. k_1 is referred to as the distinguished point of group H . It can be chosen arbitrarily, so long as it is moved by some generator of H . Several possible strategies for choosing k_1 exist. We could find the orbits of H and choose k_1 from the smallest or largest orbit (this may affect the depth of the tower); or we could simply take the first point found which is moved. In general, we use the latter technique. The group beneath $\text{Stab}(k_1)$ in the tower is $\text{Stab}(k_1, k_2)$, the stabilizer of k_2 in $\text{Stab}(k_1)$. k_2 is the distinguished point of $\text{Stab}(k_1)$. Thus, the tower has the form:

$$H \geq \text{Stab}(k_1) \geq \text{Stab}(k_1, k_2) \geq \dots \geq \text{Stab}(k_1, k_2, \dots, k_m) = \{e\}.$$

A base for H is a sequence $k_1, k_2, \dots, k_m$ of distinct points of X such that

$$H \geq \text{Stab}(k_1) \geq \text{Stab}(k_1, k_2) \geq \dots \geq \text{Stab}(k_1, k_2, \dots, k_m) = \{e\}.$$

The structure used to represent a group in the tower contains the following information:

1. a list of generators for the group.
2. the distinguished point, k , for the group.
3. the orbit of k in the group.
4. a transversal element for each coset of $\text{Stab}(k)$ in the group. The transversal element is associated with the point in the orbit of k which represents the particular coset.
5. a pointer to the next group down in the tower, $\text{Stab}(k)$.

The list of generators, the orbit of k and the transversal list will only be complete after the algorithm has finished. During the algorithm, we will have a partial list of generators, a partial set of cosets of $\text{Stab}(k)$ (i.e. a partial orbit of k in the group) and a partial transversal for $\text{Stab}(k)$.

A generator is added to a group in the tower only if it is not redundant. The redundancy check is done by the function `GroupElement` which returns a boolean result indicating whether permutation g is an element of the group H . Pascal-like pseudocode for this function follows:

```

FUNCTION GroupElement ( g: permutation; H: group ): boolean;
BEGIN
  IF g = identity permutation THEN BEGIN
    { The identity is in every group }
    GroupElement := true;
  END
ELSE BEGIN
  Let k be the distinguished point in group H;
  x := (k)g;
  IF x ∉ the orbit of k in group H THEN BEGIN
    GroupElement := false
  
```

```

END
ELSE BEGIN
  Let  $a_x$  be the transversal element associated with the point  $x$ ;
  { If  $g \in$  the group  $H$  then  $g \in \text{Stab}(k)a_x$ , or  $ga_x^{-1} \in \text{Stab}(k)$  }
  GroupElement := GroupElement(  $ga_x^{-1}$ ,  $\text{Stab}(k)$  );
END
END { GroupElement }

```

When a permutation is added to the list of generators of a group in the tower, it is applied to each of the points in the partial orbit of the distinguished point, k , to see if the orbit and the transversal of $\text{Stab}(k)$ in the group can be extended. This is done by the following procedure, AddGen, which basically does a breadth-first search of the coset diagram:

```

PROCEDURE AddGen ( g: permutation; H: group );
  { Add generator g to group H }
  BEGIN
    Add g to the list of generators of group H;
    Let k be the distinguished point in group H;
    FOR each point y in the orbit of k in H DO BEGIN
      Let  $a_y$  be the transversal element associated with point y;
       $x := (y)g$ ;
      IF  $x \notin \text{Orbit}(k)$  in group H THEN BEGIN
        { We can extend the orbit of k }
        Add x to the orbit of k;
        Product  $a_y g$  is the transversal element associated with the new point x,
          since  $(k)a_y g = (y)g = x$ ;
      END
    ELSE BEGIN
      Let  $a_x$  be the transversal element associated with the point x;
      Then  $(k)a_y ga_x^{-1} = (y)g a_x^{-1} = (x) a_x^{-1} = k$ ;
      IF NOT GroupElement(  $a_y ga_x^{-1}$ ,  $\text{Stab}(k)$  ) THEN BEGIN
        AddGen(  $a_y ga_x^{-1}$ ,  $\text{Stab}(k)$  );
      END
    END
  END

```

END
 END { FOR }

{ Now apply each generator of H to the new points added to the orbit of k by g
 in the preceding FOR loop and to any new points which are found in the
 following FOR loop }

FOR each new point y in the orbit of k DO BEGIN
 Let a_y be the transversal element associated with point y;
 FOR every generator r of H DO BEGIN
 $x := (y)r$;
 IF $x \notin$ the orbit of k in H THEN BEGIN
 Add x to the orbit of k in H;
 Product $a_y r$ is the transversal element associated with the new point x;
 END
 ELSE BEGIN
 Let a_x be the transversal element associated with x;
 Then $(k)a_y r a_x^{-1} = (y)r a_x^{-1} = (x) a_x^{-1} = k$;
 IF NOT GroupeElement($a_y r a_x^{-1}$, Stab(k)) THEN BEGIN
 AddGen($a_y r a_x^{-1}$, Stab(k));
 END
 END
 END { FOR }
 END { FOR }
 END { AddGen }

With routines GroupElement and AddGen, the code to expand a group H as a tower of stabilizers given a list of generators is:

FUNCTION Schreier(GenList: list of generators):group;
 BEGIN
 H := identity group;
 FOR each $g \in$ GenList DO BEGIN
 IF NOT GroupElement(g, H) THEN BEGIN
 AddGen(g, H);
 END;
 END;
 END;


```

    END;
    Schreier := H;
    END; { Schreier }

```

2.2 A Proof of Schreier's Algorithm

Given a list of generators for a group H , the entire orbit of the distinguished point k can be found by a breadth-first search, starting at k , of the permutation diagram. Since the points in the orbit of k are the representatives of the complete set of cosets of $\text{Stab}(k)$ in H , Schreier's algorithm will find a complete transversal for $\text{Stab}(k)$ in H .

The main part of Schreier's algorithm involves finding a set of generators for $\text{Stab}(k)$. Does Schreier's algorithm find a complete list of generators for $\text{Stab}(k)$, given a list of generators for H ? This can be answered by examining the chords in the Schreier coset diagram, $H \bmod \text{Stab}(k)$, defined by choosing a spanning tree, T .

Lemma. Let H be a group and $K \leq H$. Let T be any spanning tree of the coset diagram, $H \bmod K$. Then T defines a right transversal for K in H .

Proof. Choose coset representative e (the identity permutation) for coset K . For each vertex Ka_i of $H \bmod K$, T contains a unique path P connecting Ke to Ka_i . Then the word, $w(P)$, in the generators of H that P represents is a coset representative for Ka_i , since $Kw(P) = Ka_i$.

As Schreier's algorithm computes the orbit of k in H , a spanning tree for $H \bmod \text{Stab}(k)$ is being chosen. At the start, the tree is the single vertex, $\text{Stab}(k)$. A new edge is added to the tree when a generator maps a previously-found point in the orbit of k to a new point. Thus, the new edge does not add any cycles. Since the entire orbit of k is found by the breadth-first search, the tree will be a spanning tree at the end of the algorithm (i.e. a breadth-first spanning tree).

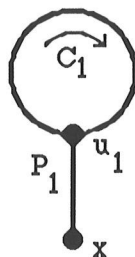
A generator for $\text{Stab}(k)$ is created for each chord found in $H \bmod \text{Stab}(k)$. If a chord from coset $\text{Stab}(k)a_x$ to coset $\text{Stab}(k)a_y$ is found while adding generator g to group H , then $a_xga_y^{-1}$ is created as a generator of $\text{Stab}(k)$. $a_xga_y^{-1}$ is, at least, an element of $\text{Stab}(k)$: $(\text{Stab}(k)a_xga_y^{-1}) = (\text{Stab}(k)a_x)ga_y^{-1} = (\text{Stab}(k)a_y)a_y^{-1} = \text{Stab}(k)$.

Any element, q , of $\text{Stab}(k)$ can be written in at least one way as a product of the generators of H : $q = b_1b_2b_3\dots b_n$, where b_i is either a generator of H or the inverse of a generator of H . $b_1b_2b_3\dots b_n$ defines a walk in $H \bmod \text{Stab}(k)$, starting at $\text{Stab}(k)$, going to $\text{Stab}(k)b_1$, to $\text{Stab}(k)b_1b_2$, ..., and finishing at $\text{Stab}(k)b_1b_2b_3\dots b_n = \text{Stab}(k)$. So q can be represented as a closed walk in $H \bmod \text{Stab}(k)$, starting and ending at $\text{Stab}(k)$, with a chosen direction of traversal.

Let C_1 be a cycle in a connected graph X . If x is any vertex of X , let P_1 be a path from x to C_1 meeting C_1 in only one vertex, as illustrated in the diagram below. In this way, every cycle of X defines a special closed walk containing exactly one cycle. The following lemma shows that every closed walk in X can be decomposed into such special closed walks based on single cycles.

Lemma. Every closed walk in a connected graph X can be written in terms of cycles.

Proof. Let W be a closed walk in X . Let x be any vertex on W . Beginning at x , follow W until the first repeated vertex, u_1 , is reached. This defines a cycle C_1 and an xu_1 -path P_1 , so that W begins as $P_1 \cdot C_1 \cdot \dots$. At this point, follow P_1^{-1} back to x .



Discard the edges $P_1 C_1 P_1^{-1}$ from W , and begin a new walk by travelling first from x to u_1 along P_1 . Continue following W until we come to a repeated vertex u_2 . Let P_2 be the xu_2 -path so defined, and C_2 the corresponding cycle. So W begins as $(P_1 \cdot C_1 \cdot P_1^{-1}) \cdot (P_2 \cdot C_2 \cdot P_2^{-1}) \cdot \dots$. Repeat until all edges of W are used up (which must occur since W is a closed walk). •

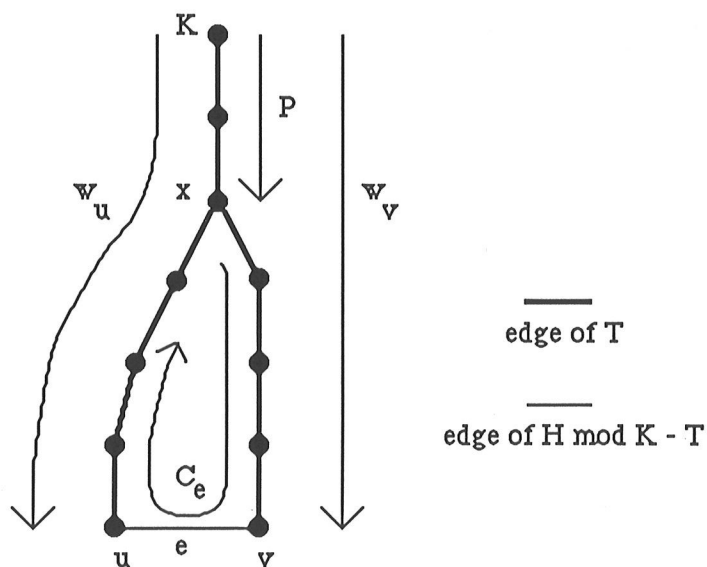
The phrase "written in terms of cycles" used in the above lemma refers to the pieces $P_i C_i P_i^{-1}$.

Let $e = uv$ be a chord of $H \bmod K$ with respect to spanning tree T corresponding to generator a of H . Define $a_e = w_u \cdot a \cdot w_v^{-1}$, where w_u and w_v are the coset representatives for u and v , respectively. Then

$w_u \cap w_v$ is a unique path P in T from K to some point x ,

$w_u - P$ is a unique path in T from x to u ,

$w_v - P$ is a unique path in T from x to v .

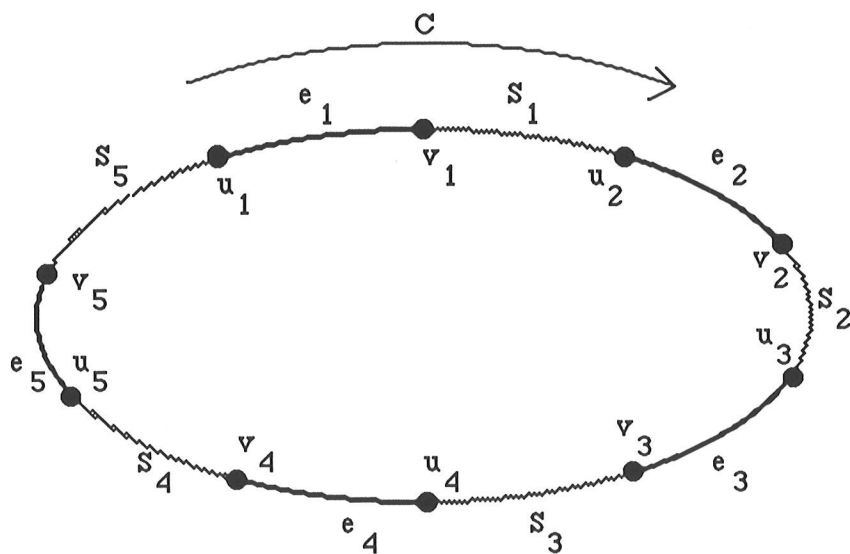


So $(w_u - P) \cdot e \cdot (w_v - P)^{-1}$ is a cycle containing one edge, e , not in T ; i.e., it is C_e , the fundamental cycle of e with respect to T . Thus, generators for K produced by Schreier's algorithm are of the form $w(P) \cdot w(C_e) \cdot w(P^{-1})$.

The following theorem shows that each piece, $(P_i \cdot C_i \cdot P_i^{-1})$, of our closed walk (i.e. of an element of $\text{Stab}(k)$) can be written as a series of walks of the form $P \cdot C_e \cdot P^{-1}$ (i.e. as a product of generators for $\text{Stab}(k)$ produced by Schreier's algorithm).

Theorem. Let X be a graph with spanning tree T . Choose a reference point x in X . Let C be any cycle in X with a direction of traversal chosen. Let C contain edges $e_1, e_2, \dots, e_n$ of $E(X) - E(T)$, in that order, and suppose that edge $e_i = u_i v_i$, in the direction of traversal. Let $P_{u(i)}$ and $P_{v(i)}$ be the unique xu_i -path and xv_i -path in T , respectively. Then $P_{u(1)} \cdot e_1 \cdot P_{v(1)}^{-1} \cdot P_{u(2)} \cdot e_2 \cdot P_{v(2)}^{-1} \cdot \dots \cdot P_{u(n)} \cdot e_n \cdot P_{v(n)}^{-1}$ defines a closed walk in X of the form $P_u \cdot C \cdot P_u^{-1}$, where u is on C and C is traversed starting at u .

Proof. Consider the cycle C . It contains $e_1, e_2, \dots, e_n$, in that order, and some edges of T . The edges $e_1, e_2, \dots, e_n$ divide C up into segments $S_1, S_2, \dots, S_n$, which are paths (possibly of length 0) belonging to T , so that $e_1 \cdot S_1 \cdot e_2 \cdot S_2 \cdot \dots \cdot e_n \cdot S_n$ is a traversal of C , beginning at u_1 .



Notice that $P_{u(i)}$ intersects segment S_{i-1} at vertex u_i , and $P_{v(i)}$ intersects segment S_i at v_i . Now $P_{u(i)} \cap S_{i-1}$ is a subset of $E(T)$, and so contains no cycle. Therefore $P_{u(i)}$ intersects S_{i-1} in a path which ends at vertex u_i . Similarly $P_{v(i)}^{-1} \cap S_i$ is a path beginning at v_i . Clearly $P_{v(i)}^{-1} \cdot P_{u(i+1)}$ is a path in T from v_i to u_{i+1} . S_i is also a path from v_i to u_{i+1} . It must be that $S_i = P_{v(i)}^{-1} \cdot P_{u(i+1)}$, since a tree T contains a unique path between any two points. Therefore $P_{u(1)} \cdot e_1 \cdot P_{v(1)}^{-1} \cdot P_{u(2)} \cdot e_2 \cdot P_{v(2)}^{-1} \cdot \dots \cdot P_{u(n)} \cdot e_n \cdot P_{v(n)}^{-1} = P_{u(1)} \cdot e_1 \cdot S_1 \cdot e_2 \cdot S_2 \cdot \dots \cdot S_{n-1} \cdot e_n \cdot P_{v(n)}^{-1}$.

But $P_{v(n)}^{-1} \cdot P_{u(1)} = S_n$, since C is a cycle, so the above expression is $P_{u(1)} \cdot e_1 \cdot S_1 \cdot e_2 \cdot S_2 \cdot \dots \cdot e_n \cdot S_n \cdot P_{u(1)}^{-1} = P_{u(1)} \cdot C \cdot P_{u(1)}^{-1}$, where C is traversed from vertex u_1 . The same argument holds for any of the u_i as starting vertex. If $P_{u(i)}$ first intersects C in segment S_k , then by beginning the traversal at vertex u_{k+1} , the above expression will reduce to $P_u \cdot C \cdot P_u^{-1}$, where u is the first vertex at which P_u meets C , and C is traversed from vertex u . •

So any $q \in \text{Stab}(k)$ can be represented as a closed walk in $H \bmod \text{Stab}(k)$. A closed walk in $H \bmod \text{Stab}(k)$ can be written in the form $(P_1 C_1 P_1^{-1}) \cdot (P_2 C_2 P_2^{-1}) \cdot \dots$. Each of

the pieces $P_i C_i P_i^{-1}$ of the closed walk can be written as a series of walks of the form $P C_e P^{-1}$. $P C_e P^{-1}$ is a generator for $\text{Stab}(k)$ produced by Schreier's algorithm. Thus, any element of $\text{Stab}(k)$ can be written in terms of the generators for $\text{Stab}(k)$ produced by Schreier's algorithm; i.e. Schreier's algorithm produces a complete set of generators for $\text{Stab}(k)$.

2.3 Complexity of Schreier's Algorithm.

Let H be a group permuting a set X . If $a, b \in H$ then $|X|$ steps are required to compute the product ab .

2.3.1 The Complexity of the Function GroupElement

GroupElement checks to see if permutation g is in group H . Point k is the distinguished point for H .

In the worst case, a call to GroupElement will require a check to see if g is the identity permutation, a check to see if $(k)g$ is in the orbit of k , the calculation of the product ga_x^{-1} , and a recursive call to GroupElement. The check to see if g is the identity permutation requires $|X|$ steps. Assume that the orbit of k is represented by a data structure of the form

VAR OrbitK: ARRAY[1..|X|] OF @TransElt;

where if $x \notin \text{orbit of } k$, OrbitK[x] is a NIL pointer. Otherwise, OrbitK[x] points at a structure representing the transversal element associated with x . In this case a check to see if $x \in \text{orbit of } k$ takes a constant amount of time.

finding the product ga_x^{-1} requires $|X|$ steps, assuming that the structure representing a_x contains a_x^{-1} . Thus we have $2|X|$ steps at each level of recursion.

At worst, there can be $|X| - 1$ stabilizers in the tower, or $|X|$ groups altogether in the tower. Hence, GroupElement can call itself at most $|X|$ times. Thus, a call to GroupElement will require $2|X|^2$ steps in the worst case.

2.3.2 The Complexity of Function Schreier

Each time a generator is added to the list of generators for group H , we go from $K \leq H$ to $K' \leq H$, and $K < K'$, since we only add a generator if it is not redundant. The smallest $[K':K]$ can be is 2. Thus, the largest number of non-redundant generators we will have is $\log_2 |H|$. $H \leq \text{Sym}(|X|)$, so $|H| \leq |X|!$.

$H \text{ mod } \text{Stab}(k_1)$ is a graph with $[H:\text{Stab}(k_1)]$ vertices, each with out-degree at most $\log_2 |H|$. The number of directed edges in the graph is (out-degree of vertices) $\cdot$ (number of vertices), or at most $(\log_2 |H|)[H:\text{Stab}(k_1)]$. The number of edges in a spanning tree of $H \text{ mod } \text{Stab}(k_1)$ is $[H:\text{Stab}(k_1)] - 1$. So, the number of chords in the Schreier coset diagram $H \text{ mod } \text{Stab}(k_1)$ is no more than $(\log_2 |H| [H:\text{Stab}(k_1)]) - ([H:\text{Stab}(k_1)] - 1) = (\log_2 |H| - 1) [H:\text{Stab}(k_1)] + 1$.

For each edge found in the spanning tree, the transversal element is found in $|X|$ steps.

For each chord found with respect to the spanning tree T , an element of $\text{Stab}(k_1)$ is found in $2|X|$ steps. Then there is a call to `GroupElement`, requiring at most $2|X|^2$ steps, and possibly a recursive call to `AddGen`. Therefore, the more chords there are, the more work is required in a call to `AddGen`.

The largest $[H:\text{Stab}(k_1)]$ can be is $|X|$; i.e. if H is transitive on X . The largest that $[\text{Stab}(k_1):\text{Stab}(k_1, k_2)]$ can be is $|X| - 1, \dots$, etc.

Therefore, in the worst case, the tower would have the form

$$H \geq \text{Stab}(k_1) \geq \text{Stab}(k_2) \geq \dots \geq \text{Stab}(k_{|X|-1}) = \{e\}, \text{ where}$$

$$[H:\text{Stab}(k_1)] = |X|,$$

$$[\text{Stab}(k_i):\text{Stab}(k_{i+1})] = |X| - i,$$

H has $\log_2 |H| \leq \log_2 (|X|!)$ generators, and $\text{Stab}(k_i)$ has $\leq \ln ((|X| - i)!)$ generators.

Suppose we were given a list of n generators completely describing H . How much work would be done by Schreier's algorithm to expand H as a tower of stabilizers?

For each of the n generators, Schreier's algorithm would call `GroupElement`, requiring at most $2|X|^2$ steps. Schreier's algorithm would make the call `AddGen(g, H)` at most $\ln |H|$ times. How much time would those calls require?

At the top level in the tower, i.e. at H , $[H:\text{Stab}(k_1)] = |X|$ transversal elements would be found. There would be $[H:\text{Stab}(K_1)] \cdot (\ln |H| - 1) + 1 = |X| \cdot (\ln |H| - 1) + 1$ chords, each giving an element of $\text{Stab}(k_1)$ in $2|X|$ steps, and causing a call to `GroupElement` requiring at most $2|X|^2$ steps.

At the level of $\text{Stab}(k_i)$ in the tower, there would be $[\text{Stab}(k_i):\text{Stab}(k_{i+1})] = |X| - i$ transversal elements and $[\text{Stab}(k_i):\text{Stab}(k_{i+1})] \cdot (\ln ((|X| - i)!) - 1) + 1$ chords.

Altogether, how many transversal elements are there?

$$\begin{aligned} \sum_{i=0}^{|X|-2} (|X| - i) &= (|X| - 1)|X| - (|X| - 2)(|X| - 3)/2 \\ &= |X|^2 - |X| - (|X|^2 - 5|X| + 6)/2 \\ &= (|X|^2 + 3|X| - 6)/2 \end{aligned}$$

Altogether, how many chords are there?

$$\begin{aligned} \sum_{i=0}^{|X|-2} (|X| - i)(\log_2 ((|X| - i)!) - 1) + 1 \\ = |X| - 1 + \sum_{j=2}^{|X|} j \log_2(j!) \\ = |X| - 1 + (\ln 2)^{-1} \sum_{j=2}^{|X|} j \ln(j!) \end{aligned}$$

Using Stirling's approximation, $j! \approx (2\pi j)^{0.5} j^j e^{-j}$, so $\ln(j!) \approx j \ln j - j + 0.5 \ln j + 0.5 \ln 2\pi$.

Thus, the number of chords will be

$$\approx |X| - 1 + (\ln 2)^{-1} \sum_{j=2}^{|X|} j(j \ln j - j + 0.5 \ln j + 0.5 \ln 2\pi)$$

$$\begin{aligned}
&= |X| - 1 + (\ln 2)^{-1} \left(\sum_{j=2}^{|X|} j^2 \ln j - [|X|(|X| - 1)(2|X| - 1)] / 6 + 1 + \right. \\
&\quad \left. 0.5 \sum_{j=2}^{|X|} j \ln j + 0.5 \ln 2\pi [|X|(|X| + 1)/2 - 1] \right)
\end{aligned}$$

Using

$$\begin{aligned}
\sum_{j=a}^b j^2 \ln j &\leq \int_a^{b+1} j^2 \ln j \, dj \\
&= j^3 (3 \ln j - 1) / 9 \Big|_a^{b+1}
\end{aligned}$$

and

$$\begin{aligned}
\sum_{j=a}^b j \ln j &\leq \int_a^{b+1} j \ln j \, dj \\
&= 0.25 j^2 (2 \ln j - 1) \Big|_a^{b+1}
\end{aligned}$$

the formula becomes

$$\begin{aligned}
&|X| - 1 + (\ln 2)^{-1} \left([(|X| + 1)^3 (3 \ln (|X| + 1) - 1) / 9 - 8(3 \ln 2 - 1) / 9] \right. \\
&\quad - [2|X|^3 + 3|X|^2 + |X|] / 6 + 1 \\
&\quad + 0.5 [0.25(|X| + 1)^2 (2 \ln (|X| + 1) - 1) - 0.25 \cdot 4(2 \ln 2 - 1)] \\
&\quad \left. + 0.5 \ln 2\pi [(|X|^2 + |X|) / 2 - 1] \right) \\
&= (|X| + 1) \ln (|X| + 1) / (3 \ln 2) + (|X| + 1)^2 \ln (|X| + 1) / (4 \ln 2) \\
&\quad - 4|X|^3 / (9 \ln 2) - (23 - 6 \ln 2\pi) |X|^2 / (24 \ln 2) \\
&\quad + (4 \ln 2 - 3 + \ln 2\pi) |X| / (4 \ln 2) - 14 / 3 \\
&\quad + (155 / 72 - 0.5 \ln 2\pi) / \ln 2 \\
&\approx 0.48(|X| + 1)^3 \ln (|X| + 1) + 0.36(|X| + 1)^2 \ln (|X| + 1) - 0.64|X|^3 \\
&\quad - 0.72|X|^2 + 0.58|X| - 2.89.
\end{aligned}$$

Therefore, the number of steps required for all calls to AddGen is less than

$$\begin{aligned}
 & (|X|^2 + 3|X| - 6) / 2 \cdot |X| + [0.48(|X| + 1)^3 \ln(|X| + 1) + 0.36(|X| + 1)^2 \ln(|X| + 1) \\
 & \quad - 0.64|X|^3 - 0.72|X|^2 + 0.58|X| - 2.89] (2|X| + 2|X|^2) \\
 & = 0.96|X|^2(|X| + 1)^3 \ln(|X| + 1) + 0.96|X|(|X| + 1)^3 \ln(|X| + 1) \\
 & \quad + 0.72|X|^2(|X| + 1)^2 \ln(|X| + 1) + 0.72|X|(|X| + 1)^2 \ln(|X| + 1) \\
 & \quad - 1.28|X|^5 - 2.72|X|^4 + 0.22|X|^3 - 3.11|X|^2 - 8.77|X|.
 \end{aligned}$$

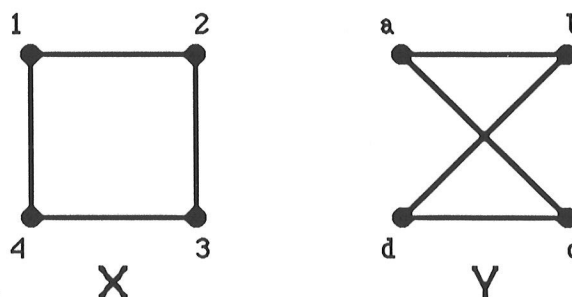
Thus the complexity of Schreier's algorithm is $O(|X|^5 \ln|X|)$. Notice that the constants involved are small.

These calculations used the pessimistic assumption that the orbit sizes and the height of the tower are as large as possible. For most groups, the number of steps will be much smaller, since only symmetric groups and alternating groups have such long towers and high transitivity.

3. Isomorphism Testing in 3-Regular Graphs

Let $X = (V(X), E(X))$ and $Y = (V(Y), E(Y))$ be graphs. An isomorphism from X to Y is a bijection $\theta : V(X) \rightarrow V(Y)$ such that $(E(X))\theta = E(Y)$.

For example,



$$E(X) = \{ 12, 23, 34, 41 \}$$

$$E(Y) = \{ ab, ac, bd, cd \}$$

Define θ by $(1)\theta = a$, $(2)\theta = c$, $(3)\theta = d$, and $(4)\theta = b$. It can easily be seen that $(E(X))\theta = E(Y)$.

The graph isomorphism problem is to determine if $X \cong Y$, given any two graphs X and Y .

We will develop an algorithm which will accept two 3-regular graphs X and Y as input and will produce either "yes" or "no" as output according to whether $X \cong Y$ or not.

A group theoretic approach to the isomorphism problem centers around the automorphism group. An automorphism of a graph X is a permutation θ of $V(X)$ such that $(E(X))\theta = E(X)$. The automorphism group, $\text{Aut}(X)$, of graph X is the set of all such automorphisms.

Note that if $\theta \in \text{Aut}(X)$ of some graph X , and $x, y \in V(X)$ such that $xy \notin E(X)$ then $(xy)\theta \notin E(X)$. Thus, $\text{Aut}(X^C) = \text{Aut}(X)$, where X^C is the complement of graph X .

Lemma. If we can find $\text{Aut}(X)$ of any graph X , then we can solve the isomorphism problem.

Proof. Assume we can find $\text{Aut}(X)$ of any graph X .

Let X and Y be two graphs. To determine if X and Y are isomorphic, do the following:

If X is connected and Y is not, or vice versa, then the two graphs are not isomorphic. Otherwise, if X and Y are not connected, replace them with their complements, which will be connected.

Create graph Z , consisting of the disjoint union of graphs X and Y . find $\text{Aut}(Z)$. If the orbit in $\text{Aut}(Z)$ of any $x \in V(X)$ contains a $y \in V(Y)$, then, since X and Y are connected, $\text{Aut}(Z)$ contains an isomorphism mapping graph X wholly onto graph Y . If this is the case, then X and Y are isomorphic. •

Let X be a graph and $e \in E(X)$. $\text{Aut}_e(X)$ is the subgroup of $\text{Aut}(X)$ fixing edge e . Thus, any element of $\text{Aut}_e(X)$ either fixes the endpoints of edge e or interchanges them.

Lemma. If we can find $\text{Aut}_e(X)$ of any graph X , we can find $\text{Aut}(X)$.

Proof. Let Y be an identical copy of X . In X , pick any edge $e \in E(X)$ and subdivide it with a new vertex x . Perform the following algorithm:

FOR each $e' \in E(Y)$ DO BEGIN

 subdivide e' with a new vertex y ;

 create a new graph Z by adding an edge xy ;

 find $\text{Aut}_{xy}(Z)$;

 IF $\text{Orbit}(x)$ in $\text{Aut}_{xy}(Z)$ contains y THEN BEGIN

 the isomorphism of X with Y mapping e to e' defines an automorphism

$\theta_{e'}$ of X mapping e to the edge of X corresponding to e' ;

$\theta_{e'}$ is a coset representative for a coset of $\text{Aut}_e(X)$ in $\text{Aut}(X)$;

 END

remove edge xy and vertex y ;

END

We now have a complete transversal of $\text{Aut}_e(X)$ in $\text{Aut}(X)$. If we also find $\text{Aut}_e(X)$, we have completely defined $\text{Aut}(X)$. •

If we want to test if two graphs X and Y are isomorphic, then use Y in place of the identical copy of X in the algorithm above, and the same algorithm will determine if X is isomorphic to Y .

3.1 Finding $\text{Aut}_e(X)$

The following technique to find $\text{Aut}_e(X)$ was presented by Furst, Hopcroft and Luks . The treatment here is similar to Hoffman [5].

Let $e = uv$. Add a new vertex, v_0 , on e . Partition $V(X)$ into $V_0 + V_1 + \dots + V_h$, where V_i is the set of vertices at distance i from vertex v_0 . Let $[V_i, V_j]$ denote the edges in X with one end in V_i and one in V_j . Let X_k denote the subgraph of X with edge set $E(X[V_0 + V_1 + \dots + V_{k-1}]) + [V_{k-1}, V_k]$, for $k \geq 1$. Let G_k be the subgroup of $\text{Aut}(X_k)$ which fixes v_0 , and let G_0 be the identity group acting on v_0 . Then $G_{h+1} = \text{Aut}_e(X)$. The algorithm of Furst, Hopcroft and Luks starts at G_0 and works up from level to level to G_{h+1} .

The difference between X_{k+1} and X_k is that X_{k+1} contains edges $[V_k, V_k]$ and $[V_k, V_{k+1}]$, which X_k does not contain. Any element of G_{k+1} induces an automorphism of X_k fixing v_0 , i.e. an element of G_k . Thus, there is a homomorphism $\pi_k: G_{k+1} \rightarrow G_k$ for each $k = 0, 1, \dots, h$. $\text{Ker}(\pi_k)$ is the set of all automorphisms of X_{k+1} which fix point-wise the sets $V_0, V_1, \dots, V_k$.

Lemma. $G_{k+1}/\text{Ker}(\pi_k)$ is isomorphic to the subgroup of G_k consisting of all automorphisms of X_k which can be extended to automorphisms of X_{k+1} .

Proof. We have $\pi_k: G_{k+1} \rightarrow G_k$ and $G_{k+1}/\text{Ker}(\pi_k) \cong \text{Im}(\pi_k)$. Every $\theta \in \text{Im}(\pi_k)$ is

an automorphism of X_k , so every such θ can be extended to an automorphism of X_{k+1} .

Algorithm. (Furst, Hopcroft and Luks)

BEGIN

partition $V(X)$ into $V_0 + V_1 + \dots + V_h$;

initialize G_0 to the identity group;

FOR $k := 0$ to h DO BEGIN

{ at this point, we know G_k . }

find $\text{Ker}(\pi_k)$, the set of automorphisms of X_{k+1} which fix $V_0 + V_1 + \dots + V_k$ point-wise;

find $G_{k+1}/\text{Ker}(\pi_k)$, automorphisms of X_{k+1} extended from those of X_k ;

together, generators for $\text{Ker}(\pi_k)$ and $G_{k+1}/\text{Ker}(\pi_k)$ define generators for G_{k+1} ;

END

{ at this point $G_{h+1} = \text{Aut}_e(X)$. }

END

3.1.1 finding $\text{Ker}(\pi_k)$

$\text{Ker}(\pi_k)$ is the set of all automorphisms in G_{k+1} which fix $V_0 + V_1 + \dots + V_k$. Thus, an element of $\text{Ker}(\pi_k)$ permutes the vertices V_{k+1} and the edges $[V_k, V_{k+1}]$; everything else is fixed. Each vertex of V_k is joined to at most two vertices of V_{k+1} . Each vertex of V_{k+1} is joined to one, two or three vertices of V_k . If x is mapped to y by an element of $\text{Ker}(\pi_k)$ then x and y must be in V_{k+1} and must be joined to the same set U of vertices in V_k . Each vertex in U must be joined only to x and y . So y must be mapped to x , and x to y . $\text{Ker}(\pi_k)$ is generated by the set of such transpositions. They can be found by sorting the vertices of V_{k+1} according to which vertices of V_k they are joined to, and then matching equivalent vertices.

3.1.2 finding $G_{k+1}/\text{Ker}(\pi_k)$

To find $G_{k+1}/\text{Ker}(\pi_k)$, we first find $\text{Im}(\pi_k)$, the set of automorphisms of G_k which

can be extended to G_{k+1} . Then the automorphisms in $\text{Im}(\pi_k)$ are extended to give $G_{k+1}/\text{Ker}(\pi_k)$.

G_k does not stabilize the edges of X within V_k , i.e. $[V_k, V_k]$. Let H_k be the action of G_k acting on V_k . Then H_k induces a permutation group $H_k^{(2)}$ acting on the set $V_k^{(2)}$ of unordered pairs of vertices in V_k . If we could stabilize sets then we could stabilize the set of edges $[V_k, V_k]$ in $V_k^{(2)}$.

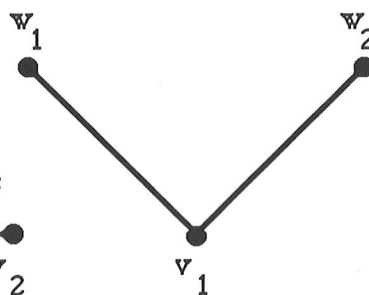
The edges $[V_k, V_{k+1}]$ must also be included. This could be accomplished in the following fashion. As noted before, each vertex in V_{k+1} can be connected to one, two or three points in V_k . So each vertex in V_{k+1} can be represented as a point in V_k , a pair of points in V_k , or a triple in V_k . Furthermore, two vertices in V_{k+1} may be connected to exactly the same set of vertices in V_k . So a point, edge or triple in V_k may represent two vertices in V_{k+1} . Such points, edges or triples are denoted as double points, edges or triples. If only one vertex in V_{k+1} is represented then the point, edge or triple is referred to as a single point, edge or triple. Notice that we now have two different sets of edges: one set represents the edges $[V_k, V_k]$ and the other represents vertices in V_{k+1} joined to only two vertices in V_k . Color the edges in the first set brown and the second set green to differentiate between the two. Here are the types of sets we wish to stabilize:

$$(\omega_i \in V_{k+1}, v_j \in V_k)$$

single points

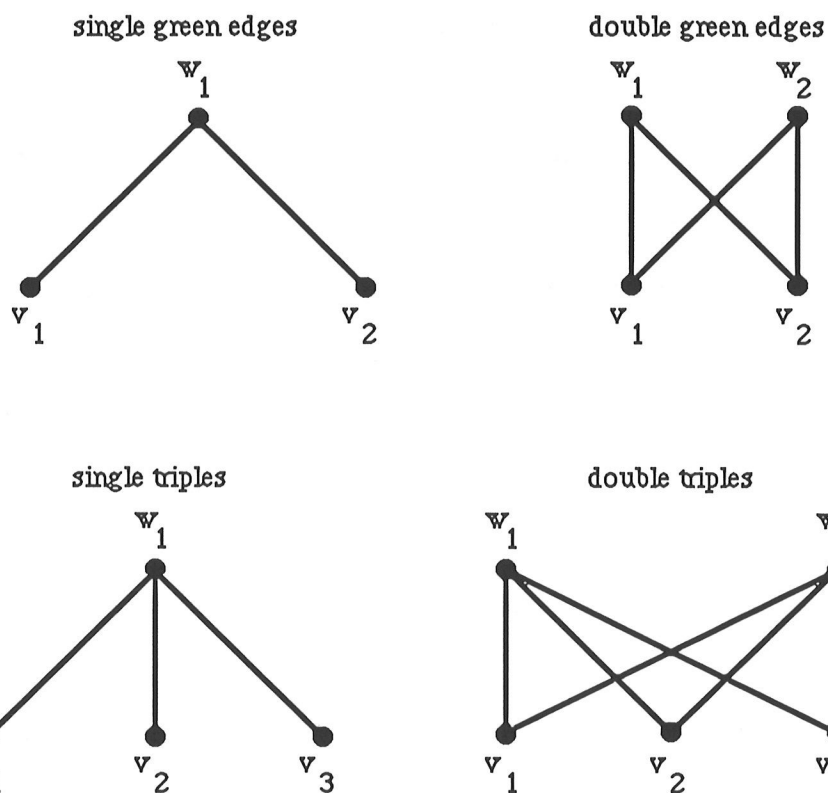


double points

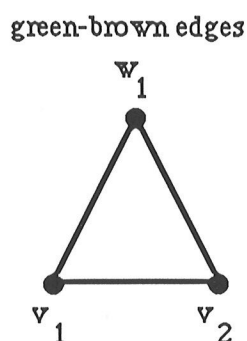


brown edges





An edge may be both a single green edge and a brown edge. These edges could be put in a separate set:



Lemma. If $\theta \in G_{k+1}$ then θ stabilizes the above sets.

Proof. Since θ stabilizes V_k , and since it is an automorphism, it must map an edge in $[V_k, V_k]$ to an edge in $[V_k, V_k]$. Thus, θ stabilizes the set of brown edges. If θ maps x

$\in V_{k+1}$ to $y \in V_{k+1}$ then, for θ to be an automorphism, the vertices in V_k to which x is connected must be mapped to the vertices in V_k to which y is connected. Also, the number of vertices in V_k to which x is connected is equal to the number of vertices in V_k to which y is connected. So θ maps triples to triples, points to points and green edges to green edges. If $w \in V_{k+1}$ is connected to exactly the same vertices in V_k as x , then w must be mapped to $z \in V_{k+1}$, $z \neq y$, where z is joined to exactly the same vertices in V_k as y . So, double points, edges or triples are mapped to double points, edges or triples, and singles are mapped to singles. Thus θ stabilizes the above sets. •

Lemma. If $\theta \in G_k$ stabilizes the above sets then θ can be extended to an element of G_{k+1} .

Proof. θ stabilizes edges of $[V_k, V_k]$ since it stabilizes the set of brown edges and the set of green-brown edges. Extend θ to θ' using the following algorithm:

BEGIN

mark each $x \in V_{k+1}$ as unprocessed;

FOR each $x \in V_{k+1}$ as yet unprocessed DO BEGIN

mark x as processed;

find the set S_x of all vertices in V_k adjacent to x ;

see if there is a $y \in V_{k+1}$ which is also adjacent to S_x in V_k ;

S_x will be a point, a green edge, a green-brown edge, or a triple (either double or single, depending on whether y exists);

θ will map S_x to S_u , a set of the same type;

S_u represents a vertex $u \in V_{k+1}$. (If y exists then S_x and S_u are doubles so

S_u would also represent $w \in V_{k+1}$.);

map x to u ;

IF y exists THEN BEGIN

mark y as used;

map y to w ;

END

END

END

Claim. $\theta' \in G_{k+1}$.

Proof. θ' stabilizes edges $[V_k, V_k]$ since θ does. Let $xu \in [V_{k+1}, V_k]$ where $x \in V_{k+1}$, $u \in V_k$. Let $(xu)\theta' = yw$, where $y \in V_{k+1}$, $w \in V_k$. The vertex u is an element of the set S_x of all vertices in V_k adjacent to x . By the above algorithm, S_x is mapped by θ and by θ' to a set S_t of the same type and x is mapped by θ' to a point y adjacent to only the vertices in S_t . Thus $w \in S_t$ and $yw \in [V_k, V_{k+1}]$. •

Hence, θ can be extended to an element of G_{k+1} . •

Suppose we are testing to see if two graphs X and Y are isomorphic, and $|V(X)| = |V(Y)| = 200$. Then graph Z , the graph created from X and Y in the algorithm to find $\text{Aut}_e(X)$, will have 402 vertices. If every vertex in V_i is adjacent to 2 vertices in V_{i+1} and no two of these vertices in V_{i+1} are the same (i.e., there is maximum fan-out from V_i to V_{i+1}), then $|V_7| = 255$. So, $|V_7^{(2)}| = 32,835$, and $|V_7^{(3)}| = 2,731,135$.

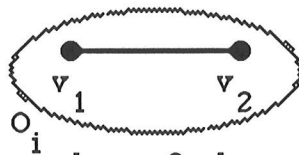
These figures are far too large for $V_7^{(2)}$ or $V_7^{(3)}$ to be handled by a computer program all at once. To break down the problem, we could examine the orbits of the vertices in V_7 . If an edge e , for instance, has one endpoint in orbit O_i and the other endpoint in orbit O_j , then e can only be mapped to another edge with one endpoint in orbit O_i and the other in O_j . We could break up the sets of points/edges/triples to be stabilized further according to which orbits the vertices are in. We would then have 16 types of sets to be stabilized.

$$(w_i \in V_{k+1}, v_i \in V_k, \text{orbits } O_1, O_2, \text{ and } O_3)$$

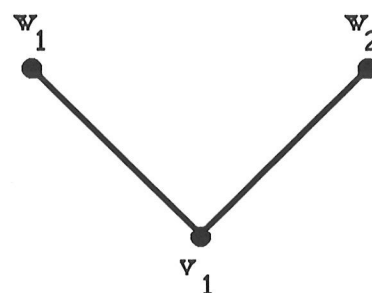
single points



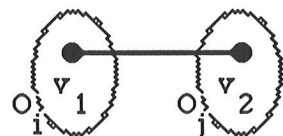
brown 1 edges



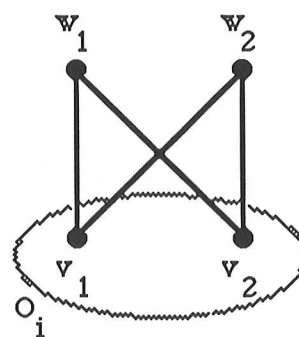
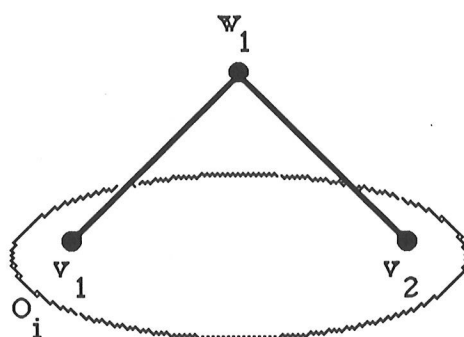
double points



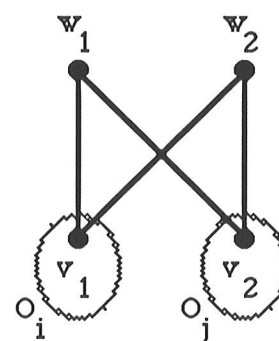
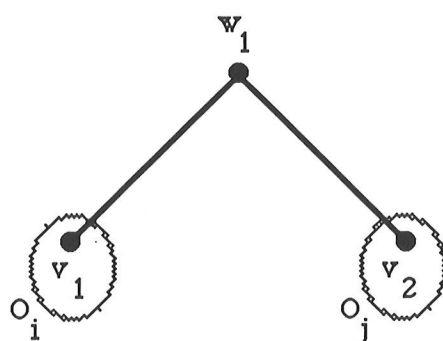
brown 2 edges



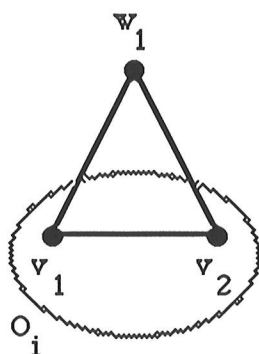
single or double green 1 edges



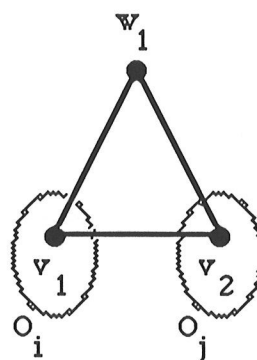
single or double green 2 edges



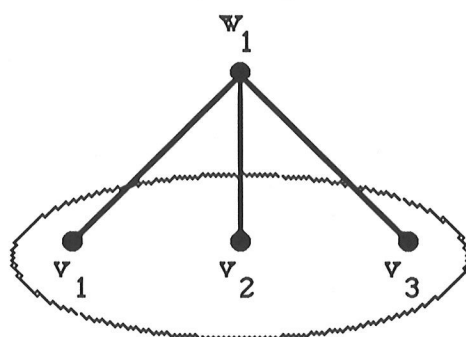
green-brown 1 edges



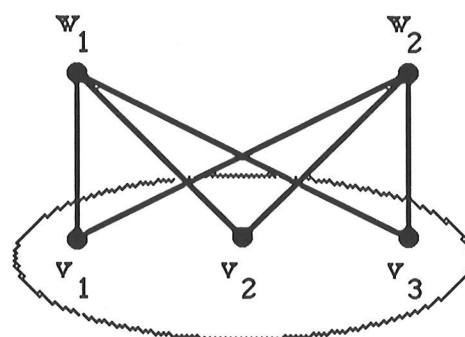
green-brown 2 edges



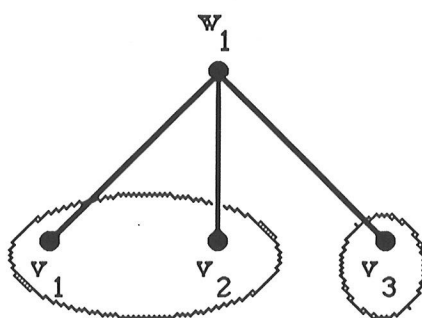
single 1 triples



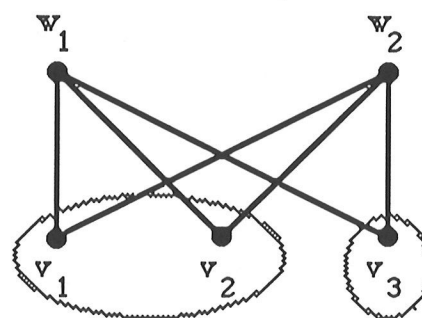
double 1 triples

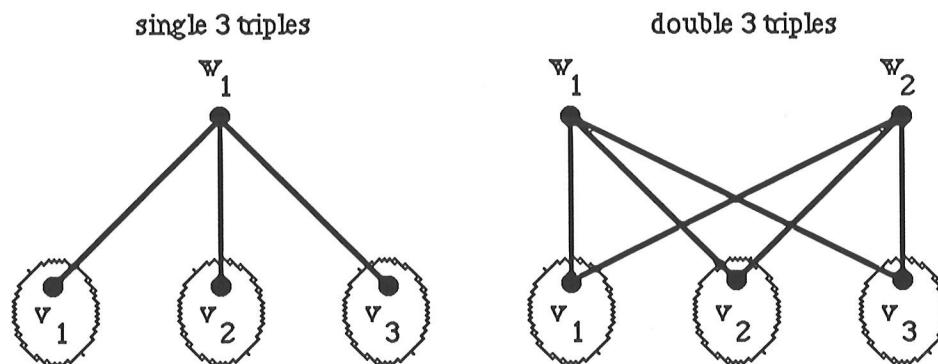


single 2 triples



double 2 triples





The following algorithm finds $\text{Ker}(\pi_k)$ and the sets which must be stabilized to find $\text{Im}(\pi_k)$.

BEGIN

mark all vertices in V_k as unused;

FOR each vertex x in V_k DO BEGIN

mark x as used;

FOR each vertex y adjacent to x in graph X_{k+1} DO BEGIN

IF $y \in V_k$ AND y has not been marked as used THEN BEGIN

{ xy is a brown edge or a green-brown edge that we have not seen before. }

find out which orbits in V_k contain x and y ;

IF $\exists q \in V_{k+1}$ adjacent to exactly x and y in V_k THEN

xy is a green-brown edge;

ELSE

xy is a brown edge;

add edge xy to a set of edges of the same type, in the same orbits in V_k as yx (or create a new set);

END; { IF $y \in V_k \dots$ }

ELSE

IF $y \in V_{k+1}$ THEN BEGIN

find the set S_y of all vertices in V_k adjacent to y ; { Note: $x \in S_y$ }

IF no vertices in S_y have been marked as used THEN BEGIN

{ we have not processed S_y before. }

IF $\exists q \in V_{k+1}, q \neq y$, adjacent exactly to the vertices in S_y in V_k THEN BEGIN

```

    the point, edge or triple,  $S_y$ , represents both  $q$  and  $y$  in  $V_{k+1}$ , so
        it is a double point, edge or triple;
    add  $(q, y)$  to  $\text{Ker}(\pi_k)$ ;
    END
ELSE
     $S_y$  is a single point, edge or triple;
    find out in which orbits in  $V_k$  contain the vertices in  $S_y$ ;
    IF  $|S_y| = 2$  AND  $S_y$  is single THEN BEGIN
         $\{S_y$  may be a green or a green-brown edge. $\}$ 
        let  $S_y = \{x, w\}$  where  $w \in V_k$ ;
        IF  $x$  is adjacent to  $w$  in graph  $X_{k+1}$  THEN
             $S_y$  is a green-brown edge;
        ELSE
             $S_y$  is a brown edge;
        END { IF  $|S_y| = 2 \dots$  }
        add  $S_y$  to a set of points/edges/triples of the same type and using the
        same orbits in  $V_k$  as  $S_y$  (or create a new set);
        END { IF no vertices in  $S_y$  have been marked }
    END { IF  $y \in V_{k+1}$  }
END { FOR each vertex  $y$  }
END { FOR each vertex  $x$  }
END

```

4. Set-Wise Stabilizers

Let G be a permutation group acting on a set X , and let M be a subset of X . We want to find $\text{Stab}(M)$, the set-wise stabilizer of M . The method used is the "divide and conquer" approach.

4.1 Intransitive Groups

If group G is intransitive on the set X then G breaks X up into several orbits $B_1, B_2, \dots, B_r$. Let $M_i = B_i \cap M$. $\text{Stab}(M)$ must set-wise stabilize each of the M_i separately.

G is transitive on B_1 . If we restrict G to B_1 by simply ignoring the other orbits, we get a group G_1 , a transitive constituent of G . $\pi_1 : G \rightarrow G_1$ is the resulting homomorphism.

If we have an algorithm which finds set-wise stabilizers in transitive groups, then we could find K_1 , the set-wise stabilizer of M_1 in B_1 , a subgroup of G_1 .

$\text{Stab}(M_1) \leq G$ is the subgroup of G such that $\text{Stab}(M_1)/\text{Ker}(\pi_1) \cong K_1$. The generators of $\text{Stab}(M_1)/\text{Ker}(\pi_1)$, together with generators of $\text{Ker}(\pi_1)$, give us $\text{Stab}(M_1)$.

$\text{Ker}(\pi_1) = \{g \in G \mid \forall x \in B_1, (x)g = x\}$. We could use Schreier's algorithm to find a set of generators for $\text{Ker}(\pi_1)$ by expanding G as a tower of stabilizers. If we use the points in B_1 as the distinguished points for the top groups in the tower, $\text{Ker}(\pi_1)$ will be one of the groups in the tower.

Let the top part of the base consisting of elements of B_1 that were used in the above expansion of G be $(b_1, b_2, \dots, b_n)$.

Once we have found generators for K_1 , we would like to expand the generators (which are of degree $|B_1|$) into elements of $\text{Stab}(M_1)/\text{Ker}(\pi_1)$ (which are of degree $|X|$). Let $k \in K_1$ and let $h \in \text{Stab}(M_1)/\text{Ker}(\pi_1)$ such that h restricted to B_1 is k . Then $(b_1)h = (b_1)k$ and h is in coset $\text{Stab}(b_1)a_1$ in H corresponding to the point $(b_1)h$. Hence, $ha_1^{-1} \in \text{Stab}(b_1)$. Then $(b_2)ha_1^{-1} = (b_2)ka_1^{-1}$ and ha_1^{-1} is in coset $\text{Stab}(b_2)a_2$ in $\text{Stab}(b_1)$ corresponding to point $(b_2)ha_1^{-1}$. Therefore, $ha_1^{-1}a_2^{-1} \in \text{Stab}(b_2)$. Continue this process

until we have $ha_1^{-1}a_2^{-1}\dots a_n^{-1} \in \text{Stab}(b_1, b_2, \dots, b_n) = \text{Ker}(\pi_1)$, or $h \in \text{Ker}(\pi_1)a_na_{n-1}\dots a_1$. Any element of $\text{Ker}(\pi_1)a_na_{n-1}\dots a_1$ can be used in place of h . In particular, $a_na_{n-1}\dots a_1$ would do. So, to find an element g which is equal to $k \in K$ when restricted to B_1 we can use the following algorithm:

```

BEGIN
  g := e;
  H := G;
  FOR i := 1 to n DO BEGIN
    bi is the distinguished point in H;
    y := (bi)k;
    let ai be the coset representative associated with point y;
    g := aig;
    k' := kai-1;
    H := Stab(bi);
  END
END

```

Once $\text{Stab}(M_1)$ is found, we need to find the stabilizer of $M - M_1$ in $\text{Stab}(M_1)$. We find the orbits of $X - B_1$ in $\text{Stab}(M_1)$. We choose an orbit B'_1 and set $M'_1 := B'_1 \cap (M - M_1)$. find $\text{Stab}(M'_1)$ in $\text{Stab}(M_1)$ in the same way that we found $\text{Stab}(M_1)$ in G . We continue this process until all of M is used up. This will give us $\text{Stab}(M)$.

Notice that we could avoid having to expand elements of K_1 to elements of $\text{Stab}(M_1)/\text{Ker}(\pi_1)$ if we simply do all calculations within the group G .

Our algorithm to find $\text{Stab}(M)$ in an intransitive group G acting on set X becomes:

```

FUNCTION SetStab( G:group; X:set; M:subset):group;
BEGIN
  H := G;
  X' := X;
  M' := M;
  WHILE M' is not empty DO BEGIN
    pick a point x ∈ X' and find its orbit B1 with a breadth-first search;
    M1 := B1 ∩ M';

```

```

IF  $|M_1| = |B_1|$  THEN
    { H already stabilizes  $B_1$ , so H already stabilizes  $M_1$  }
     $\text{Stab}(M_1) := H$ 
ELSE IF  $|M_1| = 0$  THEN { there is nothing to stabilize. }
     $\text{Stab}(M_1) := H$ ;
ELSE IF  $|M_1| = 1$  THEN BEGIN
    let  $M_1 = \{x\}$ ;
    use Schreier's algorithm to find  $\text{Stab}(M_1) = \text{Stab}(x)$ ;
    END
ELSE
    { use an algorithm for finding the set-wise stabilizer in a transitive group. }
     $\text{Stab}(M_1) := \text{TransSetStab}(H, B_1, M_1)$ ;
 $H := \text{Stab}(M_1)$ ;
 $X' := X' - B_1$ ;
 $M' := M' - M_1$ ;
END {WHILE}
SetStab := H;
END {SetStab};

```

4.2 Transitive groups

If group G is transitive on set X then we can not break the problem down using orbits in X . Fortunately, since the graphs we are testing for isomorphism are 3-regular graphs, the groups we are working with have a special property. This property allows us to develop a technique to find set-wise stabilizers in transitive graphs.

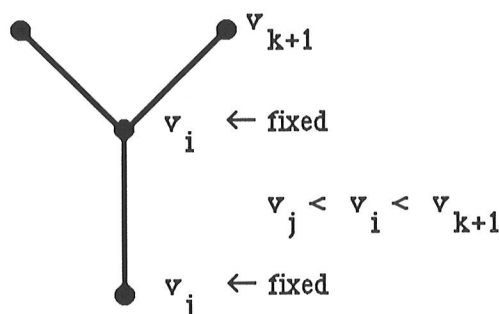
Let p be a prime number. A p -group is any group whose order is a power of p .

Tutte's Lemma. Let X be a connected 3-regular graph, and let $e \in E(X)$. Then $\text{Aut}_e(X)$ is a 2-group.

Proof. Let $e = uv$. Insert a new vertex, v_0 , on edge e . Let V_i be the subset of $V(X)$ containing the vertices at distance i from vertex v_0 . So, $V_0 = \{v_0\}$, $V_1 = \{u, v\}$, ...

Each V_i must be set-wise stabilized by $\text{Aut}_e(X)$. Order the vertices $v_1 < v_2 < \dots < v_n$ so that for all i , if $v_j \in V_i$ and $v_k \in V_{i+1}$, then $v_j < v_k$. Then each vertex v_j is joined to some vertex v_i , where $v_i < v_j$. Let $T_0 = \text{Aut}_e(X)$ and $T_i = \text{Stab}(v_0, v_1, \dots, v_i)$ (that is, each point is point-wise stabilized). Then $T_0 \geq T_1 \geq \dots \geq T_{n-1} = \{e\}$.

Consider $[T_k : T_{k+1}]$. Vertex v_{k+1} is joined to some vertex $v_i < v_{k+1}$. Vertex v_i is fixed in both T_k and T_{k+1} . The orbit of v_{k+1} in T_k is the set of all vertices in the same partition, V_p , as v_{k+1} that are also joined to v_i , since $T_k < \text{Aut}_e(X)$.



Since x is 3-regular, the orbit of v_{k+1} in T_k can contain at most 2 vertices, so $[T_k : T_{k+1}] \leq 2$. $|\text{Aut}_e(X)| = |T_0| = [T_0 : T_1] \cdot [T_1 : T_2] \cdot \dots \cdot [T_{n-2} : T_{n-1}]$ is a power of 2. •

By much the same argument, we can show that $G_k = \text{Aut}_e(X_k)$ is also a 2-group.

4.2.1 Some Properties of p-Groups

Let G be a group. Let $g \in G$. Then, $\forall h \in G$, $h^g = g^{-1}hg \in G$. By conjugating the elements of G with g , we have created a permutation, g' , of the elements of G . Here, $(h)g' = h^g$ for every $h \in G$. Let $G' = \{g' \mid g \in G\}$, the permutation representation of G in which each $g \in G$ acts by conjugation on G .

Lemma. Define $\pi : G \rightarrow G'$, where $\forall g \in G$, $(g)\pi = g'$. Then π is a homomorphism.

Proof. Let $g, h \in G$. $(gh)\pi = (gh)' \in G'$. $(g)\pi \cdot (h)\pi = g' \cdot h'$.

Let $k \in G$. $(k)(gh)' = k^g h = (gh)^{-1} k (gh) = (h^{-1} g^{-1}) k (gh) = h^{-1} (g^{-1} k g) h = (k^g)^h = ((k)g')h' = (k)(g)\pi \cdot (h)\pi$.

Since $(g)\pi \cdot (h)\pi = (gh)\pi$, π is a homomorphism. •

The center of G is $Z(G)$, the set of all elements of G which commute with all of G . Let $g \in G$. $Z(G) = g^{-1}Z(G)g = Z(G)g^{-1}g = Z(G)$. Therefore, $Z(G) \triangleleft G$.

For the homomorphism $\pi : G \rightarrow G'$, where $\pi : g \in G \rightarrow g' \in G'$,

$$\begin{aligned} \text{Ker}(\pi) &= \{g \in G \mid \forall h \in G: hg = h\} \\ &= \{g \in G \mid \forall h \in G: g^{-1}hg = h\} \\ &= \{g \in G \mid \forall h \in G: hg = gh\} \\ &= Z(G). \end{aligned}$$

$\text{Im}(\pi) = \{g' \mid g \in G\} = G'$. Since $G/\text{Ker}(\pi) \cong \text{Im}(\pi)$, we know that $G/Z(G) \cong G'$. Therefore, $G \cong G'$ if and only if $Z(G) = \{e\}$.

Let $a \in G$. The conjugacy class of a is $\{a^g \mid g \in G\}$, the set of all elements of G which can be obtained from a by conjugation. The centralizer of a is $C(a)$, the set of all elements of G which commute with a .

Consider the stabilizer of a in G' .

$$\begin{aligned} \text{Stab}(a) &= \{g' \in G' \mid (a)g' = a^g = g^{-1}ag = a\} \\ &= \{g' \in G' \mid g \in C(a)\} \\ &= (C(a))\pi. \end{aligned}$$

The orbit of a in G' is $\{(a)g' \mid g' \in G'\} = \{a^g \mid g \in G\}$, the conjugacy class of a . The size of the conjugacy class of a is $[G':\text{Stab}(a)]$.

Consider G written in terms of cosets of $C(a)$: $G = C(a)g_1 + C(a)g_2 + \dots$. Let $h \in C(a)g$. Then $h = cg$ where $c \in C(a)$. $a^h = h^{-1}ah = (cg)^{-1}a(cg) = g^{-1}c^{-1}acg = g^{-1}ac^{-1}cg = g^{-1}ag$. Therefore, $C(a)g = \{h \in G \mid a^h = a^g\}$. Hence, $[G:C(a)]$ is the size of the conjugacy

class of a .

Let $K \leq G$. The normalizer of K is $N(K) = \{g \in G \mid K^g = K\}$. Obviously $K \triangleleft N(K)$ and $N(K)$ is the largest subgroup of G in which K is normal.

Consider the action of G' on K : $(K)g' = K^g$. The set-wise stabilizer of K in G' is $\text{Stab}(K) = \{g' \in G' \mid (K)g' = K^g = K\} = (N(K))\pi$. Thus, the number of conjugates of K is $|\{K^g \mid g \in G\}| = [G:N(K)]$.

The formula $|\text{Orbit}(x)| \cdot |\text{Stab}(x)| = |G|$ now tells us that the number of conjugates of an element a or a subgroup K divides $|G|$.

4.2.2 Block Systems

Let group G be transitive on set X .

Consider $G^{(2)}$, the group induced by G acting on $X^{(2)}$, the set of unordered pairs of elements of X . Let E be an orbit of $G^{(2)}$ on $X^{(2)}$. E defines a graph $X[E]$ with vertices X and edge set E .

Example.

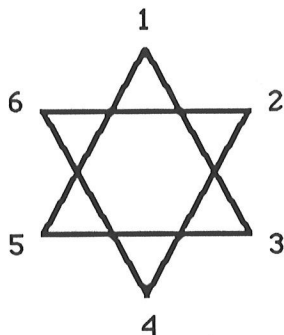
$$X = \{1, 2, 3, 4, 5, 6\}$$

$$G = \langle (1, 2, 3, 4, 5, 6) \rangle$$

G is transitive on X .

An orbit E of $G^{(2)}$ acting on $X^{(2)}$ is $E = \{\{1,3\}, \{2,4\}, \{3,5\}, \{4,6\}, \{1,5\}, \{2,6\}\}$.

$X[E]$ is



Notice that $G \leq \text{Aut}(X[E])$. $X[E]$ may have more than one component, as can be seen from the example. Since G is transitive on X , there exists a g in G which maps a vertex in one component, X_i , to a vertex in another component, X_j . Since $G \leq \text{Aut}(X[E])$, g must map component X_i wholly onto component X_j . Thus G induces a permutation of the components of $X[E]$.

Let B_i be the vertex set of the component X_i of $X[E]$. The B_i 's form a non-trivial partition of the set X such that each $g \in G$ maps each B_i wholly onto some B_j . The B_i are blocks of G , forming a block system for G .

Lemma. Let G be a transitive group acting on the set X . Let $\{B_1, B_2, \dots, B_n\}$ be a block system for G . Let K be the subgroup of G fixing each block set-wise. Then $K \triangleleft G$.

Proof. Let $g \in G$. Let B_i be one of the blocks in the above block system for G . Then $(B_i)g^{-1} = B_j$, another block in the block system for G . Let $k \in K$. Then $(B_i)k = B_i$, for every B_i , and $k^g \in K^g$. $(B_i)k^g = (B_i)g^{-1}kg = (B_j)kg = (B_j)g = B_i$. $\therefore k^g \in K$.

Let $k_1, k_2 \in K$, such that $k_1 \neq k_2$. Suppose $g^{-1}k_1g = g^{-1}k_2g$. Then $g(g^{-1}k_1g)g^{-1} = g(g^{-1}k_2g)g^{-1}$, or $k_1 = k_2$, a contradiction. So $|K| = |K^g|$. But $K \supseteq K^g$. $\therefore K^g = K$ and $K \triangleleft G$. •

Lemma. Let G be a transitive group acting on set X . G has a block system if and only

if $G^{(2)}$ has a disconnected orbit E on $X^{(2)}$.

Proof. We have already seen that if $G^{(2)}$ has a disconnected orbit E then the vertex sets of the components of $X[E]$ form a block system for G .

Let G have a block system $\{B_1, B_2, \dots, B_n\}$. Consider the graph $X[E]$ created by the orbit E of $\{x, y\} \in B_1^{(2)}$. Since each $g \in G$ maps B_1 wholly onto another block B_i , no $g^{(2)} \in G^{(2)}$ can map $\{x, y\}$ onto $\{u, v\}$ where $u \in B_i$ and $v \in B_j \neq B_i$. Thus $X[E]$ has no connections between the vertices in B_i and the vertices in B_j . $\therefore E$ is a disconnected orbit of $G^{(2)}$. •

A permutation group G is primitive if every orbit of $G^{(2)}$ on $X^{(2)}$ is connected. Otherwise it is imprimitive. If G is imprimitive then a block system for G is also called a system of imprimitivity.

4.2.3 p-Groups and Block Systems

Let G be a transitive group acting on a set X , with a block system $\{B_1, B_2, \dots, B_n\}$. Let B be any block and $x \in B$.

Lemma. $\text{Stab}(x) \leq \text{Stab}(B) \leq G$.

Proof. Since each $g \in G$ maps B wholly onto another block and each $h \in \text{Stab}(x)$ maps x onto x , each $h \in \text{Stab}(x)$ maps B onto B , i.e. $h \in \text{Stab}(B)$. •

Lemma. The size of each block is $[\text{Stab}(B) : \text{Stab}(x)]$.

Proof. Since G is transitive, all blocks have the same size.

$$\text{Stab}(B) = \text{Stab}(x)a_1 + \text{Stab}(x)a_2 + \dots + \text{Stab}(x)a_k.$$

$\text{Stab}(x)a_i$ consists of all permutations of G mapping to $(x)a_i \in B$. Thus $\{(x)a_i \mid i = 1, 2, \dots, k\}$ must be all of B . $\therefore |B| = [\text{Stab}(B) : \text{Stab}(x)]$. •

Lemma. The number of blocks is $[G:\text{Stab}(B)]$.

Proof. $G = \text{Stab}(B)b_1 + \text{Stab}(B)b_2 + \dots + \text{Stab}(B)b_m$. $\text{Stab}(B)b_i$ consists of all permutations in G mapping block B to block $(B)b_i$. Again, since G is transitive on X , $\{(B)b_i \mid i = 1 \dots m\}$ must be all the blocks. $\therefore$ the number of blocks is $[G:\text{Stab}(B)]$. •

Theorem. Let G be transitive on X . Let $x \in X$. If G contains a subgroup H between $\text{Stab}(x)$ and G , then G is imprimitive and the orbit of x in H is a block for G .

Proof. Let $\text{Stab}(x) < H < G$ and let B be the orbit of x in H . Since $\text{Stab}(x) \neq H$, B contains more than x . $G = Hb_1 + Hb_2 + \dots + Hb_k$. Hb_i consists of all permutations mapping B to $(B)b_i$.

Let $g \in G$. Then $g \in Hb_i$ and $g = hb_i$ where $h \in H$. $((B)b_j)g = (B)b_jhb_i$. But $b_jhb_i \in G$, so $b_jhb_i \in Hb_s$. Every permutation in Hb_s maps B to $(B)b_s$. $\therefore ((B)b_j)g = (B)b_s$. $\therefore \{(B)b_1, (B)b_2, \dots, (B)b_k\}$ is a block system for G . •

Corollary. A transitive group G is imprimitive if and only if there is some subgroup between G and a vertex stabilizer.

Theorem. Let K be an intransitive normal subgroup of the transitive permutation group G . If $K \neq \{e\}$, then the orbits of K form a block system for G .

Proof. If K is intransitive, let $B_1, B_2, \dots, B_k$ be the orbits of K . G permutes the orbits of a normal subgroup. Since G is transitive, all B_i have the same size, and they form a block system for G . •

Lemma. The center of a p -group has order p^k , for some $k \geq 1$.

Proof. $Z(G) \leq G$, so $|Z(G)|$ divides $|G| = p^r$. $\therefore |Z(G)| = p^k$ for $k \geq 0$. Recall G' acting on G : $g' \in G'$ maps $h \in G$ to $(h)g' = hg' = g^{-1}hg$. The orbits of G' are the

conjugacy classes of G . The size of each orbit must divide $|G|$. $\{e\}$, where e is the identity of G , is an orbit of size one. Let $O(p^i)$ be the set of all orbits of size p^i . Then

$$|G| = p^r = 1 \cdot |O(1)| + p \cdot |O(p)| + \dots + p^r \cdot |O(p^r)|.$$

Now, p divides p^r , and p divides $(p \cdot |O(p)| + \dots + p^r \cdot |O(p^r)|)$. $\therefore p$ divides $|O(1)|$. But $\{g\}$ is an orbit only if $g \in Z(G)$. $\therefore |Z(G)|$ is divisible by p and $Z(G) \geq 1$. Hence, $|Z(G)| = p^k$ for some $k \geq 1$. •

Lemma. Let G be a p -group, and let P be a subgroup of index p . Then $P \triangleleft G$.

Proof. Consider $G \bmod P$, a transitive permutation group of degree p acting on the cosets of P . $G \bmod P$ is a subgroup of the symmetric group of degree p , which has order $p!$.

Let $\theta : g \in G \rightarrow g^* \in G \bmod P$, and $|G| = p^r$. $\text{Ker}(\theta) \leq G$. $\therefore \text{Ker}(\theta) = p^i$ for some $0 \leq i < r$. $G/\text{Ker}(\theta) \cong \text{Im}(\theta) = G \bmod P$. So $|G \bmod P| = p^k$ where $k > 0$. But $|G \bmod P|$ divides $p!$ and the only power of p greater than 1 dividing $p!$ is p . So $|G \bmod P| = p$. Since $G \bmod P \cong G/H$, where H is the largest subgroup in P which is normal in G , and $[G:P] = |G/H| = p$, it follows that $P = H$, and $P \triangleleft G$. •

Theorem. Let G be a group of order p^r . If K is a subgroup of G of order p^k where $k < r$, then K is contained in a subgroup $P \leq G$ of order p^{k+1} .

Proof. By induction on r .

If $r = 1$, then $|G| = p$, $K = \{e\}$ and the result is clearly true.

Consider $r > 1$. Assume the hypothesis is true for $|G| = p^s$, where $s < r$. If $k = r - 1$, then $P = G$ is the subgroup of order p^{k+1} . So consider $k < r - 1$. If $K \triangleleft G$, then $|G/K| = |G \bmod K| = p^{r-k} < p^r$. By the induction hypothesis, the identity subgroup of G/K is contained in a subgroup P/K of order p . Then $P \leq G$ and $|P| = p^{k+1}$.

If K is not normal, allow G' to act on K . Let $X = \{K_1, K_2, \dots, K_t\}$ be the set of all

conjugates of K . Then $t \neq 1$, since K is not normal. Since the size of the orbit must divide $|G'| = |G/Z(G)|$, a power of p , t is a power of p .

Consider K' , the subgroup of G' corresponding to K , and the action of K' on X . K' is a p -group, so the size of each orbit of X in K' is either one or a power of p . K' fixes K , so that K is an orbit of size one.

There must be several orbits of size one. Suppose that K_i is also an orbit of size one. Then $K' \leq \text{Stab}(K_i)$, so that $K \leq N(K_i)$, the normalizer of K_i . Since $K \neq K_i \leq N(K_i)$, $|N(K_i)| > |K|$. But $N(K_i) \neq G$, since K is not normal in G . So, $|N(K_i)| = p^q$, where $q < r$. Then by the induction hypothesis, K is contained in a subgroup P of $N(K_i)$ of order p^{k+1} , so that $K \leq P \leq N(K_i) \leq G$ as required. •

Given a p -group G , we can start at the identity subgroup $\{e\}$ and, by repeatedly applying the above theorem, we can build a tower of subgroups $\{e\} = P_r \diamond P_{r-1} \diamond \dots \diamond P_1 \diamond P_0 = G$, where $[P_i: P_{i+1}] = p$.

Theorem. A transitive permutation group G of order p^r is primitive if $r = 1$ and imprimitive if $r > 1$.

Proof. Let G act on X and let $x \in X$. Since G is transitive, $|X| = p^k$, for some $k \geq 1$.

If $k = 1$, then since $G \leq \text{Sym}(X)$, the order of G is p and $|\text{Stab}(x)| = 1$. There can be no subgroup between $\text{Stab}(x)$ and G .

If $k > 1$, then $|\text{Stab}(x)| = p^{r-k}$. By the previous theorem, $\text{Stab}(x)$ is contained in a subgroup P of order p^{r-1} , so G is imprimitive. The number of blocks in the corresponding block system is $[G:P] = p$ and the size of each block is p^{k-1} . •

Our technique for finding a set-wise stabilizer of subset M in a transitive p -group G is to find an intransitive subgroup $P \diamond G$ fixing the blocks in a block system for G with p

blocks. In this case $[G:P] = p$. We will then find the set-wise stabilizer of M in P , which is intransitive, and then search the cosets of P . (Notice that there are as few cosets as possible.)

4.2.4 finding a Block System in a Transitive p -Group

Suppose we want to find a block system for the transitive p -group G acting on set X .

What we do is to use a merge-find data-structure[1] to find the orbits of $G^{(2)}$ acting on $X^{(2)}$. For each orbit of $G^{(2)}$, we then use another merge-find data structure to see if the orbit is disconnected.

If the disconnected orbit we find has more than p blocks and we wanted exactly p blocks, we could iterate the above procedure. Let the set of blocks found be $Y = \{B_1, B_2, \dots, B_s\}$, where $s = p^i$ and $i > 1$. Then G induces a permutation group G_Y acting on the block system Y . If $g \in G$, let g_Y be the corresponding permutation in G_Y . We could use a merge-find data structure to find the orbits of $G_Y^{(2)}$ acting on $Y^{(2)}$ and then use another merge-find data structure to find a disconnected orbit.

Since G is a p -group, it has a block system with p blocks. If block system Y has more than p blocks then, since G_Y is a transitive p -group, G_Y has a block system and so $G_Y^{(2)}$ has a disconnected orbit.

Suppose we iterate and find that $Y^{(2)}$ has a disconnected orbit, i.e. the orbit of $\{B_i, B_j\}$ is disconnected.

Claim. Let $x \in B_i$ and $u \in B_j$. The orbit of $\{x, u\}$ in $X^{(2)}$ is equivalent to the orbit of $\{B_i, B_j\}$ in $Y^{(2)}$.

Proof. Let $\{B_m, B_n\}$ be in the orbit of $\{B_i, B_j\}$. $\exists g_Y^{(2)} \in G_Y^{(2)}$ mapping $\{B_i, B_j\}$ to $\{B_m, B_n\}$. WLOG, assume g_Y maps B_i to B_m and B_j to B_n . Then $g \in G$ maps $x \in B_i$ to some $x' \in B_m$ and $u \in B_j$ to some $u' \in B_n$.

If $h \in G$ maps $x \in B_i$ to $x' \in B_m$ and $u \in B_j$ to $u' \in B_n$, then $\{x', u'\}$ is in the orbit of $\{x, u\}$ and $h_y \in G_y$ maps B_i to B_m and B_j to B_n , so $\{B_m, B_n\}$ is in the orbit of $\{B_i, B_j\}$ in $Y^{(2)}$. •

Thus, once we have found the orbits in $X^{(2)}$, we have found the orbits in $Y^{(2)}$, where Y is any block system we choose from the disconnected orbits of $X^{(2)}$.

Our algorithm, using $p=2$, becomes:

BEGIN

queue $Q := \text{empty}$;

use the merge-find algorithm to find the orbits of $X^{(2)}$;

$E := \text{first orbit of } X^{(2)}$;

WHILE (the no. of components $\neq 2$) AND ($E \neq \text{NIL}$) DO BEGIN

set up the merge-find data structure A used to find the blocks so that each $x \in X$
is in its own block;

use the merge-find algorithm to find the components of $X[E]$;

IF the no. of components $\neq 2$ THEN BEGIN

IF there are more than 2 component and less than $|X|$ components THEN

put E on queue Q ;

$E := \text{next orbit of } X^{(2)}$;

END

END { WHILE }

$E := \text{first orbit in } Q$;

WHILE there are more than 2 components DO BEGIN

choose any $\{x, y\} \in E$;

IF x and y are not in the same component THEN BEGIN

create a merge-find structure B to find blocks of the current blocks;

use the merge-find algorithm with E to find the blocks of blocks;

IF there are fewer blocks than there were, but still more than 1 THEN

apply E to the merge-find structure A ;

END { IF x and $y \dots$ }

$E := \text{next orbit in } Q$;

END {WHILE }

{ merge-find data structure A now contains 2 blocks. }

END

Once we have found a block system containing p blocks, we want to find $P = \text{Stab}(B)$, where B is one of the blocks. We can do this using Schreier's algorithm and the merge-find data structure, which maps each point in X to the block containing the point. We will create a tower $G \geq \text{Stab}(B) \geq \text{Stab}(x_1) \geq \dots \geq \text{Stab}(x_n)$, where $x_i \in X$. Each coset of $\text{Stab}(B)$ in G is represented by a block in the block system, so the group G can not be represented by the group data structure in chapter 1. To expand the transversal of $\text{Stab}(B)$ in G , we apply each of the generators to any point in B and then check which block contains the resulting point using the merge-find data structure. All other groups in the tower can be created using the algorithms in chapter 1.

4.2.5 An Algorithm for finding Set-Wise Stabilizers in Transitive p -Groups.

Let G be a transitive p -group acting on the set X . Let $\{B_1, B_2, \dots, B_p\}$ be a block system for G , and let $P \trianglelefteq G$ be the subgroup of index p fixing the sets B_i . Write $G = P + Pa + Pa^2 + \dots + Pa^{p-1}$, where $a \in G$. Let M be a subset of X . Let P_M denote the set-wise stabilizer of M in P .

$$\begin{aligned} P_M &= \{h \in P \mid (M)h = M\} \\ &= \{h \in P \mid h \in \text{Stab}(M) \leq G\} \\ &= P \cap \text{Stab}(M), \text{ a subgroup of } \text{Stab}(M). \end{aligned}$$

G/P has order $|G|/|P| = p$. So G/P is a cyclic group of order p . Pick any $b \in G-P$. G/P is generated by Pb . In particular, if some coset $Pa \neq P$ in G contains a b fixing M , then Pb generates G/P .

Thus, for $b \in G-P$ such that $(M)b = M$:

$$G = P + Pb + Pb^2 + \dots + Pb^{p-1}.$$

$$\text{Stab}(M) = (P + Pb + Pb^2 + \dots + Pb^{p-1}) \cap \text{Stab}(M)$$

$$= P \cap \text{Stab}(M) + Pb \cap \text{Stab}(M) + Pb^2 \cap \text{Stab}(M) + \dots + Pb^{p-1} \cap \text{Stab}(M).$$

Now, $Pb^i \cap \text{Stab}(M)$ is a coset of P_M in G and b^i is in that coset, so we get

$$\text{Stab}(M) = P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}.$$

Claim. $P_M \triangleleft \text{Stab}(M)$.

Proof. $\forall g \in \text{Stab}(M)$, P_M^g is a subset of $P^g = P$. P_M^g fixes M since g fixes M . $P_M = \text{Stab}(M) \cap P$. Therefore, $P_M^g = P_M$.

This gives us

Theorem. Let G be a transitive p -group with a block system $\{B_1, B_2, \dots, B_p\}$. Let $P \triangleleft G$ be the subgroup of index p fixing the block B set-wise. Let $P_M = P \cap \text{Stab}(M)$. Then either $\text{Stab}(M) = P_M$ or else Pa , where $a \notin P$, contains an element b fixing M and $\text{Stab}(M) = P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}$.

Thus our algorithm for finding a set-wise stabilizer in G becomes:

FUNCTION TranSetStab(G :group; X :set; M :subset):group;

{ G is a transitive p -group acting on X . We want $\text{Stab}(M)$.}

BEGIN

find an intransitive subgroup P , of index p ;

{If $|G| = p$, set $P =$ identity subgroup and let each point of X be a "block" to allow the recursion to work. Otherwise P fixes a block system. }

$P_M := \text{SetStab}(P, X, M)$;

{Either P_M is all of $\text{Stab}(M)$, or $\text{Stab}(M) = P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}$, for some $b \in Pa$, where $(M)b = M$ }

choose a permutation $a \notin P$ to represent coset Pa ;

$h := \text{CosetStab}(P, X, M, (M)a^{-1})$;

{search P for h mapping M to $(M)a^{-1}$. $(M)ha = (M)a^{-1}a = M$, so $ha \in Pa$ is the permutation b that we wanted.}

```

IF h ≠ NIL THEN
    PM := <PM, ha>;
    TransSetStab := PM;
END; {TranSetStab}

```

4.2.6 The Function CosetStab

We would like to find a $g \in P$ mapping M to N where P is an intransitive p -group acting on set X and M and N are subsets of X . The orbits of P are the blocks $\{B_1, B_2, \dots, B_p\}$ of G .

Consider block B_i . Let $M_i = M \cap B_i$ and $N_i = N \cap B_i$. If P contains a g mapping M to N , then g maps M_i to N_i . Thus we employ a divide-and-conquer technique much like the one we used to find a set-wise stabilizer in an intransitive group G .

We first find a g_1 mapping M_1 to N_1 . The set of all such g_1 is the coset $\text{Stab}(M_1)g_1$ in P . We now want a $g \in \text{Stab}(M_1)$ mapping M to $(N)g_1^{-1}$. $\text{Stab}(M_1)$ creates orbits in $X - B_1$. We choose an orbit B_2 and let $M_2 = B_2 \cap M$. Now we search $\text{Stab}(M_1)$ for g_2 mapping M_2 to $N_2 = B_2 \cap (N)g_1^{-1}$. If there is such a g_2 then $(M_1)g_2g_1 = (M_1)g_1 = N_1$, and $(M_2)g_2g_1 = (B_2 \cap (N)g_1^{-1})g_1 = (B_2)g_1 \cap N$.

We continue this process until all of M is used up, or until, at stage i , we find that no such g_i exists. In the latter case, we know that there is no $g \in P$ mapping M to N so we return a nil pointer. Otherwise, we eventually have $g = g_k g_{k-1} \dots g_2 g_1$ mapping M to N , so we return g .

```

FUNCTION CosetStab( P:group; X:set; M,N:subsets):permutation;
{Search P for an element g mapping M to N. P is intransitive on X.
  If g exists, return g; otherwise return nil.}
BEGIN
  H := P;
  X' := X;
  M' := M
  g := e;
  WHILE (g ≠ nil) AND ( |M'| > 0 ) DO BEGIN
    find an orbit  $B_i$  of H on X';
     $M_i := M' \cap B_i$ ;
     $N_i := (N)g^{-1} \cap B_i$ ;
    { we want  $g_i$  mapping  $M_i$  to  $N_i$ . }
    IF  $|M_i| \neq |N_i|$  THEN
      {no such  $g_i$  exists. return NIL.}
      g := NIL;
    ELSE
      IF  $|M_i| = 1$  THEN BEGIN
        use a breadth-first search to search the coset diagram for  $g_i$  mapping  $M_i$ 
        to  $N_i$  (if there is such a  $g_i$ .);
        IF  $g_i \neq \text{NIL}$  THEN BEGIN
          {the set of all such  $g_i$  is a coset of  $\text{Stab}(M_i)$ , the point-wise stabilizer
            of  $M_i = \{x\}$ .}
          H :=  $\text{Stab}(M_i)$ ;
          g :=  $g_i \cdot g$ ;
          X' := X' -  $B_i$ ;
          M' := M' -  $M_i$ ;
          END {IF  $g_i \neq \text{NIL}$  }
        ELSE
          g := NIL;
        END
      ELSE BEGIN {  $|M_i| \geq 2$  }
        { H is transitive on  $B_i$ . See if H restricted to  $B_i$  contains a  $g_i$  mapping
           $M_i$  to  $N_i$ .}
         $g_i := \text{TranCosetStab}(H, B_i, M_i, N_i)$ ;

```

```

    IF  $g_i \neq \text{NIL}$  THEN BEGIN
        {The set of such  $g_i$  is a coset of  $\text{Stab}(M_i)$  in  $H$ .}
         $H := \text{TranSetStab}(H, B_i, M_i);$ 
         $g := g_i \cdot g;$ 
         $X' := X' - B_i;$ 
         $M' := M' - M_i;$ 
        END { IF  $g_i \neq \text{NIL}$  }
    ELSE
         $g := \text{NIL};$ 
    END
END { WHILE }
CosetStab :=  $g;$ 
END {CosetStab}

```

4.2.7 Function TranCosetStab

We wish to find an element g in the transitive p -group G acting on X , such that g maps M to N , where M and N are subsets of X .

Let $P \triangleleft G$ be an intransitive p -group of index p . $G = P + Pc + Pc^2 + \dots + Pc^{p-1}$, where $c \in G$ and $c \notin P$. Let $P_M = \text{Stab}(M) \cap P$. Then $P = P_M a_1 + P_M a_2 + P_M a_3 + \dots + P_M a_k$. We know that $\text{Stab}(M)$ in G is either P_M or $P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}$, for some $b \notin P$.

Suppose $\text{Stab}(M) = P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}$, where $b \notin P$. Furthermore, suppose coset Pc^i contains some $g = hc^i$, where h is an element of P , mapping M to N . Then $(M)P_M b^j g = (M)g = N$. Consider $Pc^j = Pc^{j-i+i} = (Pc^{j-i})c^i = (c^{j-i}P)c^i = c^{j-i}(Pc^i) = c^{j-i}(Pg) = (Pc^{j-i})g$.

If Pc^{j-i} contains something fixing M then Pc^j contains something mapping M to N . $b \in G$, so $b \in Pc^t$ or $b = dc^t$ where $d \in P$. Consider $Pb^x = P(dc^t)^x = Pc^{tx}$. For what x do we have $tx \equiv j - i \pmod{p}$? Since p is a prime and $0 \leq t < p$, $\gcd(t, p) = 1$ and there is a solution y to the above congruence.

Therefore $P_M b^y$ is a subset of Pc^j , so Pc^j contains something mapping M to N .

Thus, if $\text{Stab}(M) = P_M + P_M b + P_M b^2 + \dots + P_M b^{p-1}$ and any coset Pc^i contains a g mapping M to N , then every coset Pc^j contains something mapping M to N . In this case, we need only examine P itself to find such a g , if there is one.

If $\text{Stab}(M) = P_M$ and $g \in G$ maps M to N then $(M)P_M g = (M)g = N$. $P_M g$ is a subset of some coset Pc^i . We must search each of the cosets Pc^i for such a g .

FUNCTION TranCosetStab(G:group; X:set; M,N:subsets):permutation;

{G is a p-group transitive on the set X. Look for a permutation $g \in G$ such that g maps M to N . If there is such a g , return it. Otherwise, return NIL.}

BEGIN

find an intransitive subgroup $P \triangleleft G$ of index p ;

{If $|G| = p$, we can take $P = \text{identity group}$. Otherwise, P will fix a block system.}

$g := \text{CosetStab}(P, X, M, N)$;

IF $g = \text{NIL}$ THEN BEGIN

{Either $\text{Stab}(M) = P_M$ and some other coset of P may contain the g we want, or $\text{Stab}(M) = P_M + P_M b + \dots + P_M b^{p-1}$ and no other coset of P contains such a g (since P doesn't).}

choose a permutation $c \in G$ such that $c \notin P$.

$b := \text{CosetStab}(P, X, M, (M)c^{-1})$;

IF $b = \text{NIL}$ THEN BEGIN

{ $\text{Stab}(M) = P_M$ and some coset Pc^i may contain the g we want.}

$k := 1$;

WHILE $(k \leq p - 1)$ AND $(g = \text{NIL})$ DO

$g := \text{CosetStab}(P, X, M, (N)c^{-k})$;

END { IF $b = \text{NIL}$ }

END { IF $g = \text{NIL}$ }

TranCosetStab := g ;

END {TranCosetStab}

5. The Complexity of the Algorithm

5.1 The Complexity of Finding $\text{Aut}_e(X)$.

Given a 3-regular graph X with n vertices, and $e \in E(X)$, how many steps are required to find $\text{Aut}_e(X)$? To answer this, we find the complexity of the algorithm at the end of section 3.1.

What is the order of G_i ? $G_i \leq \text{Sym}(n)$ and G_i is a 2-group. Therefore $|G_i|$ is at most the largest power of 2 that divides $n!$. What is the largest power of 2 dividing $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$? Every second number in $n!$ is divisible by 2, giving us $\lfloor n/2 \rfloor$ powers of 2. Every fourth number is divisible by a second 2, giving $\lfloor n/4 \rfloor$ more powers of 2. Every eighth number is divisible by a third 2, giving us $\lfloor n/8 \rfloor$ more powers of 2. Continue this process until we reach the $2^{\lfloor \log_2 n \rfloor}$ th number, which provides one last power of 2. Thus we have in total

$$\begin{aligned}
 \sum_{i=1}^{\lfloor \log_2 n \rfloor} \lfloor n/2^i \rfloor &\leq \sum_{i=1}^{\lfloor \log_2 n \rfloor} n/2^i \\
 &= n \cdot \sum_{i=1}^{\lfloor \log_2 n \rfloor} 1/2^i \\
 &= n [2 - 1/(2^{\lfloor \log_2 n \rfloor} - 1)] \\
 &\leq 2n - 4
 \end{aligned}$$

Thus, $|G_i| \leq 2^{2n-4}$. If we use Schreier's algorithm to restrict the number of generators used to represent G_i , we will need at most $2n - 4$ generators.

Let n_i be the number of vertices at distance i from the vertex v_0 (which is added on edge e), i.e. the number of vertices in the set V_i .

Each vertex x in V_i can be adjacent to at most 2 other vertices in V_i , since x is adjacent to at least one vertex in V_{i-1} . Thus, $|[V_i, V_i]| \leq n_i$.

Each vertex in V_i can be adjacent to at most 2 vertices in V_{i+1} , and each vertex in V_{i+1} is adjacent to at most 3 vertices in V_i . Thus $||V_i, V_{i+1}|| \leq \min(2n_i, 3n_{i+1})$.

Assuming we have a data structure representing V_i which allows us to check if $x \in V_i$ in constant time, we can partition the vertices into $V_0 + V_1 + V_2 + \dots + V_h$ in $3n$ steps.

To find the orbits of vertices in V_k , we use a tree implementation of the merge-find algorithm which merges smaller trees into larger trees and performs path compression. Using this implementation, merges require constant time and n_k finds require $O(n_k \cdot \alpha(n_k))$ time, where $\alpha(n_k)$ is a function which is not constant but grows more slowly than $\log n_k$ [1]. Thus we will apply each generator g of G_k to each vertex x in V_k , finding the roots of the merge-find trees representing the orbits of x and $(x)g$, and (possibly) merging the two trees. Thus, we will perform at most $2(2n - 4) \cdot n_k$ finds, requiring $O(n \cdot n_k \cdot \alpha(n_k))$ steps.

The algorithm which finds $\text{Ker}(\pi_k)$ and the sets which must be stabilized to find $\text{Im}(\pi_k)$ is given at the end of section 3.1.2.

The first step, to "mark all vertices in V_k as unused", requires n_k steps.

The outermost FOR loop, "FOR each vertex x in V_k DO BEGIN", loops n_k times. Inside the loop, we examine each vertex y adjacent to x , or at most 3 vertices.

If $y \in V_k$, we find the roots of the merge-find trees representing the orbits of x and y . We also search the three vertices adjacent to y to find a q in V_{k+1} adjacent to exactly x and y in V_k , which requires $3 \cdot 3$ steps. We then look for a set containing edges of the same type as xy using the same orbits as x and y . This can require at most $2 \cdot n_k \cdot (n_k - 1) / 2$ comparisons (if each edge is in its own set). Thus if $y \in V_k$, we require 2 finds and $9 + 2 \cdot n_k \cdot (n_k - 1) / 2$ steps.

If $y \in V_{k+1}$, we find the set S_y of all vertices in V_k adjacent to y in 3 steps. We then search for another vertex $q \in V_{k+1}$, adjacent to exactly the vertices in S_y in V_k , in 3^3 steps. Adding (q, y) to $\text{Ker}(\pi_k)$ requires n steps. Then we find the roots of the merge-find trees of the elements in S_y , requiring at most 3 finds. To check if x is adjacent to w , if S_y

$= \{x, w\}$, requires 3 steps. To add S_y to a set of n -tuples of the same type as S_y and using the same orbits requires at most $3 \cdot n_k \cdot (n_k - 1)$ steps (if all n -tuples are triples, each in a different set). Thus, if $y \in V_{k+1}$, we require 3 finds and $3 \cdot n_k \cdot (n_k - 1)(n_k - 2) / 6 + 3 + 27 + n + 3$ steps, or 3 finds and $3 \cdot n_k \cdot (n_k - 1)(n_k - 2) / 6 + 33 + n$ steps. So, the algorithm in section 3.1.2 needs at most $n_k + n_k \cdot 3 \cdot (3 + 3 \cdot n_k \cdot (n_k - 1)(n_k - 2) / 6 + 33 + n)$ steps. $9n_k$ finds would require $O(n_k \cdot \alpha(n_k))$ steps. Thus, the complexity of the algorithm in section 3.1.2 is $O(n_k^4 + n \cdot n_k)$.

For the moment, let us ignore the next step, which is to stabilize all the sets of n -tuples to find $\text{Im}(\pi_k)$.

We next extend generators of $\text{Im}(\pi_k)$ to generators of $G_{k+1}/\text{Ker}(\pi_k)$ using the first algorithm in section 3.1.2. n_{k+1} steps are required to mark each vertex in V_{k+1} as processed. The FOR loop repeats n_{k+1} times. Within the FOR loop, 3 steps are required to find S_x , 3^3 steps are required to find a $y \in V_{k+1}$ also adjacent to S_x , 3 steps are required to map S_x and 3^3 steps are required to find u and w . Thus, extending each generator of $\text{Im}(\pi_k)$ to a generator of $G_{k+1}/\text{Ker}(\pi_k)$ requires $n_{k+1} + 60n_{k+1} = 61n_{k+1}$ steps. If the number of generators used to represent $\text{Im}(\pi_k)$ is $2n - 4$, then this step requires $O(n \cdot n_{k+1})$ time.

The total time required for the algorithm at the end of section 3.1 is

$$\begin{aligned}
 & 3n + \sum_{i=0}^h [O(n \cdot n_k \cdot \alpha(n_k)) + O(n_k^4 + n \cdot n_k) + O(n \cdot n_{k+1})] \\
 & = 3n + \sum_{i=0}^h O(n \cdot n_k \cdot \alpha(n_k) + n_k^4 + n \cdot n_k)
 \end{aligned}$$

plus the time required to stabilize all the sets of n -tuples at each level. We know that

$$\sum_{i=0}^h n_k = n,$$

and that

$$\alpha(n_k) \leq \log n_k \leq \log n.$$

Thus,

$$\sum_{i=0}^h n_k \cdot \alpha(n_k) \leq \sum_{i=0}^h n_k \log n = n \log n.$$

Also,

$$\sum_{i=0}^h n_k^4 \leq \left(\sum_{i=0}^h n_k \right)^4 = n^4$$

and

$$\sum_{i=0}^h n \cdot n_k = n^2.$$

Therefore, the total time required is at most $O(n^4 + n^2 \log n)$, plus the time required to stabilize the sets of n -tuples.

5.2 The Complexity of Finding a Block System.

Suppose G is a p -group transitive on a set X , where $|G| = p^k$ and $|X| = x$. What is the complexity of finding a block system for G on X ?

The complexity will depend on the size of the generating set used to represent G . If we use Schreier's algorithm to reduce the number of generators, we will start with the identity group and add a generator only if it is not redundant (i.e. only if it allows us to expand the

current subgroup of G to a larger subgroup) until we have all of G . Since G is a p -group, adding a generator must expand the subgroup by at least a factor of p . Thus, we will add at most $\log_p p^k = k$ generators.

To find a block system we use a merge-find algorithm. The implementation used is a tree implementation with path compression, which merges the smaller tree into the larger. A merge requires constant time. If we are working with a set of size n , n finds require $O(n \cdot \alpha(n))$ time, where $\alpha(n)$ is a function that is not constant but grows very slowly [1]. A single find can require no more than $\log n$ steps.

To find a block system, we first merge-find the orbits of pairs of points in X . For each pair $\{a, b\}$ in $X^{(2)}$ and each generator g of G , we find the roots of the merge-find trees for $\{a, b\}$ and for $(\{a, b\})g$ and (possibly) merge the two trees together. This will require $2 \cdot k \cdot O(C_{x,2} \cdot \alpha(C_{x,2}))$ steps, where $C_{n,r}$ denotes " n choose r ".

The next step, in the worst case, requires us to find the components of each orbit of $X^{(2)}$. For each element $\{a, b\}$ of an orbit, we find the roots of the merge-find trees associated with a and b . The number of finds for each orbit is twice the size of the orbit. Thus the total number of finds is $2 \cdot C_{x,2}$, requiring $2 \cdot C_{x,2} \cdot O(\log x)$ time.

If we are forced to iterate the previous step, we will, at worst, apply each orbit of pairs to blocks of blocks and then again to the merge-find trees representing the components (if the orbit did not reduce down to one block). Thus, the total procedure requires

$$\begin{aligned} & 2 \cdot k \cdot O(C_{x,2} \cdot \alpha(C_{x,2})) + 3 \cdot 2 \cdot C_{x,2} \cdot O(\log x) \\ &= O(k \cdot C_{x,2} \cdot \alpha(C_{x,2}) + C_{x,2} \cdot \log x) \\ &= O(x^2 \log x + k \cdot x^2 \cdot \alpha(x^2)) \text{ steps.} \end{aligned}$$

5.3 The Complexity of Set-wise Stabilisation.

In the set-wise stabilisation algorithm of Chapter 4, G is a p -group acting on X . We wish to find $\text{Stab}(M)$. If G is intransitive, the problem breaks down into at most $|X|$

subproblems, so we consider the transitive case. Here we find a subgroup P of index p , stabilising a block system of G . We first find the set-wise stabiliser P_M within P , which is intransitive. Then either $\text{Stab}(M)=P_M$ or else a coset Pa contains b fixing M , and $\text{Stab}(M)=P_M+P_Mb+P_Mb^2+\dots+P_Mb^{p-1}$. Searching Pa for such a b is equivalent to searching P for an element mapping M to $N=(M)a^{-1}$. The algorithm uses a recursive call to find P_M and then chooses one coset Pa and examines P for b mapping M to N (when $p=2$ there is no saving over searching all p cosets, but when $p>2$, this will be faster). So we want the complexity of searching P for b mapping M to N . The set-wise stabilisation problem reduces to this in general, if we consider the case $M=N$.

Consider now the general problem of searching a group G for an element mapping M to N . If G is intransitive, we first restrict M and N to the first orbit, getting M_1 and N_1 , and look for an element mapping M_1 to N_1 . The set of all such elements is a coset of $\text{Stab}(M_1)$. Having found such an element, we now work within $\text{Stab}(M_1)$, and consider the second orbit, etc. If G is transitive, we find P of index p fixing a block system as before, and examine the p cosets of P in G until we find the element b desired. P is intransitive. Examining any coset of P requires examining the constituents of P acting on the p blocks of G . So searching G requires searching p^2 similar groups, each a factor of p or more smaller.

The recursion will break G down into a block system and cosets of P , which will in turn be broken down similarly until in the worst case we are working with a subgroup of order 1. So it would seem that we may have to examine every element of G in the worst case. But since the block systems successively partition X finer and finer, the maximum depth of the recursion is $\log_p |X|$, at which point the block partition of X is discrete and M and N are also partitioned discretely. At this point there are only p elements to check to find the b desired. So at most $p^{\log_p |X|} = |X|$ elements of G must be examined in the worst case to find generators for $\text{Stab}(M)$. The number of block systems which must be

constructed is at most $1+p+p^2+\dots+p^{\log|X|-1} = (|X|-1)/(p-1) \approx |X|/p$. Each requires $O(x^2 \log x)$ steps, giving $O(x^3 \log x)$ steps in total for constructing block systems and searching G for generators of $\text{Stab}(M)$.

The set-wise stabilisation procedures are called for several different colours of edges, but the number of colours is a constant. In the actual implementation used, the sets to be stabilised are broken down as much as possible (see Chapter 6), but presumably, this requires less work than calling the procedures exactly once for each colour, without breaking down the sets. So the number of calls is $O(h)$, where h is the number of levels in the graph. Since $h < n$, the number of vertices of G , this adds one power of n to the complexity. Finally, the stabilisation is acting on pairs and triples of vertices, so that in the worst case $|X| = O(n^3)$, although this must be very rare, if it occurs at all. This gives a complexity of $O(n^7 \log n)$ for the entire algorithm.

6. The Program

This chapter contains a description of the data structures used in the program and a discussion of how the algorithm was implemented.

Since the program uses arrays whose dimensions are calculated at execution time, we would like to be able to allocate an array of any size we choose during the execution of the program. Standard Pascal does not provide execution-time dimensioning. Using the PVS compiler, this facility can be provided in a relatively simple fashion. Storage for variables in a Pascal program is allocated by calling an assembly-language routine `ampxnew`. `Ampxnew` is passed the number of bytes to be allocated and returns a pointer to an area of the given size. This routine can be called to allocate a structure at execution time as long as the number of bytes required for the structure is known and `ampxnew` is declared in the following fashion: `FUNCTION ampxnew( size: integer ): pointer; EXTERNAL;`

PVS provides a type pointer which is a generic pointer, i.e. it can point at any type of structure. Thus, if `ampxnew` returns a generic pointer, the routine can be used to allocate any type of structure.

Example: The following code allocates an array `B` of integers which contains a number of elements given by the variable `BSize`:

CONST

HugeSize = 1000; { The max. # of elements an array can contain. }

TYPE

PtrIntArray = @IntArray;

IntArray = array[1..HugeSize] of integer;

VAR

BSize: integer;

B: PtrIntArray; { A pointer to the array to be allocated. }

FUNCTION ampxnew(size: integer): pointer; EXTERNAL;

BEGIN

... { Calculate the number of elements we would like the array to contain. }

B := ampxnew(BSize * 4);

{ B now points at an array of BSize integers. }

{ To access element `i` of the array: }

B@[i] := 5;

...
END;

Unfortunately, the Pascal debugging routine which checks indices at execution time will not be very helpful for arrays which were allocated at execution time. As far as the index checking routine is concerned, B points at an array of size HugeSize. Thus, the statement $B@[\text{HugeSize}] := 5;$ will not be flagged as an error by the index checking routine if an array of size less than HugeSize has been allocated. The programmer must take extra care to ensure that he does not attempt to access elements for which no storage was allocated.

Since each vertex of the graph Z for which we are finding $\text{Aut}_e(Z)$ is adjacent to only three other vertices, the data structure representing the graph Z contains an adjacency list for each vertex. The data structure type Graph is actually a pointer to an array. The array is allocated at execution time. Each element of the array points at a structure representing a vertex which contains the adjacency list for that vertex.

The partition V_i consists of all vertices at distance i from the vertex v_0 . V_i is represented by exchanging pointers in the array Z so that $|V_i|$ contiguous elements of the array point at the records representing the vertices in V_i . Thus, we need to store the indices in array Z of the first and last vertices in V_i . Since the record representing vertex i will not necessarily be pointed at by element i in array Z, we will also need array IndexMap which is a mapping from vertices to their indices in array Z. Thus, $Z@[\text{IndexMap}@[i]]$ points at the record representing vertex i.

The type permutation is actually a pointer to an array which contains the permutation. The array is allocated at execution time.

The data structure representing a group is much as described in chapter 2. The records representing the transversal elements are linked together so that the orbit of the distinguished point can be processed without examining the points which are not in the orbit. The following are the type declarations for the group structure:

```

GenName = 'A' .. 'Z';
pt = 1..HugeSize;
permutation = @permtyp;

PtrGenElt = @GenElt;
GenElt = record          { generator structure }
    Gen: permutation;
    GenInverse: permutation;
    GenSymbol: GenName; { used for printing purposes }
    NextGenElt: PtrGenElt { next generator structure in list }
end;

PtrTransElt = @ TransElt;
OrbitArray = @OrbitArrayType;
OrbitArrayType = array[ pt ] of PtrTransElt;
TransElt = record        { coset representative (transversal element) }
    a: permutation;
    aInverse: permutation;
    NextTransElt: integer
end;

TransRec = record        { indices of the 1st and last transversal elements in a
                           linked list of transversal elements }
    TransStart: integer;
    TransEnd: integer
end;

Group = @GroupType;
GroupType = record          { Structure representing a group }
    Degree: pt;
    Order: integer;
    Gens: PtrGenElt;        { a pointer to a linked list of generators }
    Point: pt;              { the distinguished point of the group }
    Orbit: OrbitArray;      { the orbit of point }
    Trans: TransRec;        { linked list of transversal elements }
end;

```

```

Stab: Group          { a pointer to a structure representing
                      Stab(Point) }
end;

```

The merge-find algorithm is used in various places in the program. The smaller merge-find tree is always merged into the larger and path compression is used to speed up the program [1]. The merge-find data structures consist of a mapping from the objects to be merged to their merge-find pointers (an array of pointers allocated at execution time). The structure containing the merge-find pointer is a variant record. It contains the pointer to its parent in the merge-find tree if it is not the root of the tree. If it is the root of the tree, it contains the number of elements in the tree. If we want to access all elements in a particular tree then the root of each tree also contains a pointer to a circularly linked list of elements in the tree, which is updated every time the tree is merged with another.

When we are examining edges $[V_i, V_i]$ and $[V_i, V_{i+1}]$ to find the sets of n -tuples which must be stabilized, we create linked lists of structures representing sets of n -tuples: one list for points, one for brown-one edges, one for brown-two edges, and so on. Each structure representing a set of n -tuples contains a pointer to a linked list of the n -tuples in the set, the type of n -tuples, the size of the n -tuples, the number of n -tuples in the set, whether the n -tuples are single or double, and how many n -tuples are possible of that particular type using the same orbits as the n -tuples in the set. After all sets of n -tuples are found, the sets are reorganized to reflect the amount of work each would take to set-wise stabilize. In some cases we can use Schreier's algorithm to find the set-wise stabilizer. If a set contains one n -tuple and the points in the n -tuple are each in a different orbit, then the set-wise stabilizer of the set is the subgroup which point-wise stabilizes each of the points in the n -tuple. Thus, if we use Schreier's algorithm with the points of the n -tuple as the first points in the base used, we can pick the required subgroup out of the tower of stabilizers produced. If a set contains one n -tuple of size three where two of the points are

in the same orbit and the third is in a different orbit, then the set-wise stabilizer of the set must point-wise stabilize the third point and set-wise stabilize the first two points. Thus, to find the set-wise stabilizer in this case, we first call Schreier's algorithm to find the point-wise stabilizer of the third point. The group we want is a subgroup of the point-wise stabilizer. The point-wise stabilizer may have different orbits than the original group. In particular, the first two points may now be in different orbits so that Schreier's algorithm can be used again to find the set-wise stabilizer of the first two points. If the first two points are still in the same orbit then the routine TranSetStab must be used to find the set-wise stabilizer of the two points. For sets which must be stabilized using the SetStab routines, if the total number of n -tuples possible of the same type using the same orbits as the n -tuples in set A is m and the total number of n -tuples possible of the same type using the same orbits as the n -tuples in set B is more than m , then it would probably require more work to stabilize set B than set A. So we order such sets in ascending order according to the total number of such n -tuples possible.

We always try to choose the set of n -tuples that would require the least amount of work to stabilize. Once we have stabilized a set of n -tuples, the orbits of the points may be broken into smaller orbits and some of the sets of n -tuples may be broken into smaller sets. These changes may cause the ordering of the sets to be changed. However, rather than going through all the sets and reordering them, an approximation scheme is used to find the next set which would require a lesser amount of work to stabilize. The set which was the second best choice when we chose the previous set will probably still be close to the head of the list if we reorganized all sets. If we reorganized that set and chose the subset requiring the least amount of work to stabilize, we would probably have a good approximation to the best choice and will have saved ourselves a great deal of work.

If we choose a set M that must be stabilized using the SetStab routines, we must pass both M and a set X of n -tuples that contains M and is already set-wise stabilized (we use

the set of all possible n -tuples of the same type as the n -tuples in M and using the same orbits). Within the SetStab routines the sets of n -tuples are stored as arrays of n -tuples. Each n -tuple is sorted so that if the n -tuple is stored as $(a_1, a_2, \dots, a_n)$ then $a_1 < a_2 < \dots < a_n$. The n -tuples within the array are sorted so that if $(a_1, a_2, \dots, a_n)$ comes before $(b_1, b_2, \dots, b_n)$ then $a_1 = b_1, a_2 = b_2, \dots, a_i < b_i$ for some $1 < i \leq n$. Thus, when we want to find the intersection of two sets in SetStab or CosetStab (for example, to find $(N)g^{-1} \cap B_i$) we can search one set for elements of the other more quickly.

To find a group $P \triangleleft G$ that fixes a block system for G , we use the method described in section 4.2.4. We produce a tower of stabilizers with G at the top and P under G . For group G , the distinguished 'point' is one of the two blocks in the block system. The orbit contains only the two blocks. The transversal element associated with the distinguished block is the identity. Thus, we can dispense with using the usual structure used to represent an orbit and store only the transversal element for the second block. This orbit of two blocks is processed using a special version of Schreier's algorithm which manipulates blocks and n -tuples instead of points. Group P , the next group down in the tower, and all groups beneath it in the tower are created using the regular Schreier's algorithm which manipulates points. Thus, P is represented as a normal tower of stabilizers. This representation is very useful in routines TranSetStab and TranCosetStab when we try to find $b \in G$ such that $b \notin P$. All we need do in this case is to test each generator of G for membership in P using the routine GroupElement. One of the generators will not be a member of P since $P < G$.

In routine CosetStab, we try to find g in group G mapping set M_i onto set N_i . If M_i and N_i each contain only one n -tuple, say $M_i = \{m\}$ and $N_i = \{n\}$, then we search the coset diagram of $G \bmod \text{Stab}(m)$ to find g mapping m to n . M_i is contained in the orbit B_i of n -tuples. We allocate an array of pointers to transversal elements to represent the orbit B_i , set the transversal element of m to be the identity permutation, and then perform a

breadth-first search to find the transversal element associated with n . We then find $\text{Stab}(M_i)$ which is $\text{Stab}(m)$. $\text{Stab}(m)$ can be found using Schreier's algorithm. The distinguished 'point' associated with group G , the top group in the tower, is not a point, but is an n -tuple m , and the orbit of m is an orbit of n -tuples. $\text{Stab}(m)$, the next group down from G , and all stabilizers beneath it in the tower can be created using the standard Schreier's algorithm working with points, but a special routine must be written to work with group G at the top of the tower since n -tuples are manipulated instead of points.

We include a listing of the main procedures used, omitting input and output procedures, procedures which dispose of structures which are no longer useful, and other such procedures in order to save space. Because the algorithm to test if $X \cong Y$ joins them by an edge, which is then stabilized as $\text{Aut}_e(Z)$, where $Z = X + Y$, our program is a program to compute edge-stabilizers. The listing of the program follows the bibliography.

We also include a listing of some sample output with the program. The graph in question is the Heawood graph [2] with one edge fixed. The execution time for this graph was 1.72 seconds on an Amdahl 5850 computer. A similar calculation for this graph using the graph isomorphism program of [8] required 0.07 seconds. This program also produces a graph certificate.

In conclusion, it seems that the group-theoretic algorithms for graph isomorphism are primarily of a theoretical interest. The present algorithm is restricted to graphs with maximum degree 3. Although this is a polynomial algorithm, the combinatorial algorithm of [8] seems to perform better. Although there are a number of places where the program could have been made more efficient, it would still be only for graphs with maximum degree 3. The generalization to larger, fixed degree complicates the problem enormously.

Bibliography

1. A. Aho, J. Hopcroft, J. Ullman, Data Structures and Algorithms, Addison-Wesley, Don Mills, Ontario, (1983).
2. J. A. Bondy, U. S. R. Murty, Graph Theory with Applications, Elsevier North Holland, New York, (1976).
3. M. Furst, J.E. Hopcroft, E. Luks, "A subexponential algorithm for trivalent graph isomorphism", Tech. Rept. 80-426, Dept. of Comp. Sci., Cornell University, (1980).
4. M. Hall, jr., The Theory of Groups, MacMillan, New York, (1959).
5. C. M. Hoffmann, Group Theoretic Algorithms and Graph Isomorphism, Springer-Verlag, Berlin, (1982).
6. W. L. Kocay, "Group Algorithms and Graph Isomorphism", course notes, University of Manitoba, (1984).
7. Charles C. Sims, "Computation with permutation groups", A.C.M. Second Symposium on Symbolic and Algebraic Manipulation, L.A., California, Ed. S.R. Petrick, (1971).
8. W.L. Kocay, "Abstract data types and graph isomorphism", J.C.I.S.S., to appear.

* Graph Z *

```

1, 2 }
1, 14 }
1, 6 }
2, 3 }
2, 11 }
3, 4 }
3, 8 }
4, 5 }
4, 13 }
5, 6 }
5, 10 }
6, 7 }
7, 8 }
7, 12 }
8, 9 }
9, 10 }
9, 14 }
10, 11 }
11, 12 }
12, 13 }
13, 14 }

```

*** The Group G_2 ***

Order: 8

Generators:

(3 11)
(1 2)(3 14)(6 11)

*** The Group G_3 ***

Order: 128

Generators:

(10 12)
(1 2)(3 14)(4 9)(5 10)(6 11)(7 12)(8 13)
(3 11)(4 10)(8 12)

*** The Subgroup of Aut(Z) Fixing Edge (1, 2) ***

Order: 16

The Group:

The order of the group: 16

generators:

B: (1 2)(3 6)(4 5)(7 8)(9 12)(10 13)(11 14)
A: (3 11)(4 10)(8 12)(9 13)

orbit of 1:

1 e
2 (1 2)(3 6)(4 5)(7 8)(9 12)(10 13)(11 14)

Stabiliser of 1:

The order of the group: 8

generators:

B: (5 13)(6 14)(7 9)(10 12)
A: (3 11)(4 10)(8 12)(9 13)

orbit of 2:

2 e

Stabiliser of 2:

The order of the group: 8

generators:

B: (5 13)(6 14)(7 9)(10 12)
A: (3 11)(4 10)(8 12)(9 13)

orbit of 3:

3 e
11 (3 11)(4 10)(8 12)(9 13)

Stabiliser of 3:

The order of the group: 4

generators:

B: (4 8)(5 9)(6 14)(7 13)
A: (5 13)(6 14)(7 9)(10 12)

orbit of 4:

4 e
8 (4 8)(5 9)(6 14)(7 13)

Stabiliser of 4:

The order of the group: 2

generators:

A: (5 13)(6 14)(7 9)(10 12)

orbit of 5:

5 e
13 (5 13)(6 14)(7 9)(10 12)

Stabiliser of 5:

*** END OF GROUP ***

```

*****
MAINLINE PROGRAM
*
* This program reads in two 3-regular graphs X and Y. One edge
* from each graph has been chosen. The program tests to see if
* there is an isomorphism from X to Y mapping the chosen edge e in
* graph X to the chosen edge e' in graph Y.
* The check is done by creating a new graph Z from the graphs
* X and Y in the following manner:
* - a new vertex v0 is added to graph X on edge e (this vertex
*   becomes vertex 1 in graph Z).
* - a new vertex v0' is added to graph Y on edge e' (this vertex
*   becomes vertex 2 in graph Z).
* - a new edge e" is added from v0 to v0'
* The subgroup of the automorphism group of the graph Z that fixes
* edge e" (i.e. either fixes or swaps v0 and v0') is found. If
* the subgroup contains an automorphism mapping v0 to v0' then
* there is an isomorphism from X to Y, mapping edge e to edge e'.
*
* Representation of the graph Z:
* Type Graph:
* - an array of pointers to adjacency lists (one list for
*   each vertex in the graph).
* - allocated at execution time.
* Type GraphElt:
* - a structure representing one vertex in the graph
* - contains the vertex, and list of the vertices adjacent to
*   this vertex (there will be 3 adjacent vertices since all
*   graphs tested are 3-regular).
*
* Representation of subgroup:
* The subgroup found is represented as a tower of point
* stabilizers. Each group in the tower is represented by the
* data structure Group. While the subgroup is being found,
* intermediate groups are often represented as just a list of
* generators (i.e. just a linked list of GenElt structures).
*
* Permutation - permutations are in vector form
*               - the type permutation is a pointer to an
*                 execution-time dimensioned array
*
* Group - a structure representing a group as a stabiliser of
*          a point that the permutations act upon and a
*          transversal for that stabiliser
*          - Degree - the number of points being permuted
*          - Order - the order of the group
*          - Gens - the generators of the group
*                  - a pointer to a linked list of generators
*                    (GenElt's)
*          - Point - the point for which the stabiliser is found
*                    (and the Orbit) within the group
*          - Orbit - the Orbit of Point within the group

```

```

*
* - a pointer to an execution-time dimensioned
*   array of pointers to transversal elements
*   (TransElt's). If Orbit@[i] is nil then point
*   i is not in Orb(Point) otherwise Orbit@[i]
*   points at the coset representative
*   corresponding to point i
* - Trans - a structure used to link elements in the
*           orbit together in a queue
*           - TransStart - index of the first point in the
*             orbit
*           - TransEnd - index to the last point in the queue
*                     of orbit elements
* - Stab - the stabiliser of Point
*         - pointer to another group structure
*
* GenElt - a structure representing a generator
*          - Gen - a pointer to the generator
*          - GenInverse - a pointer to the inverse of the
*                       generator
*          - GenSymbol - the letter name given to the generator
*          - NextGenElt - pointer to the next GenElt structure in
*                        the list
*
* TransElt - a structure representing a transversal element
*            - a, aInverse - pointers to the transversal element
*                          and its inverse
*            - NextTransElt - the index of the next transversal
*                          element in the queue

```

```

*****
PROGRAM main;
CONST
  hugesize = 10000;
TYPE
  { GRAPH }
  Graph = @GraphType;
  GraphEltPtr = @GraphElt;
  GraphType = array[ 1..hugesize ] of GraphEltPtr;
  GraphElt = RECORD
    { 1 record/vertex in graph }
    { contains info about a vertex }
    in the graph Z
    vertex: integer;
    adjvertex: array[ 1..3 ] of integer
    { the vertices to which this
      vertex is adjacent }
  END;

  { GROUP (ORBITS ARE POINTS ONLY) }

```

```

GenName = 'A'..'Z';          { generator symbol representation }
pt = 1..HugeSize;
permutation = @permtyp;
permtyp = array[ pt ] of pt;

PtrGenElt = @GenElt;
GenElt = record
    { generator structure }
    Gen: permutation;
    GenInverse: permutation; { generator inverse }
    GenSymbol: GenName;      { name of generator }
    NextGenElt: PtrGenElt
end;

PtrTransElt = @TransElt;
OrbitArray = @OrbitArrayType;
OrbitArrayType = array[ pt ] of PtrTransElt;
TransElt = record
    { coset representative }
    a: permutation;
    aInverse: permutation;
    NextTransElt: integer
end;

TransRec = record
    TransStart: integer;
    TransEnd: integer
end;

Group = @GroupType;
GroupType = record
    Degree: pt;
    Order: integer;
    Gens: PtrGenElt;
    Point: pt;
    Orbit: OrbitArray;
    Trans: TransRec;
    Stab: Group
end;

VAR
    G: Group;

DEF
    Z: Graph;
    ZSize: integer;

FUNCTION ampxnew( size: integer ): pointer; EXTERNAL;

{ THE MAIN PROGRAM HAS NOT BEEN INCLUDED HERE SINCE IT CONSISTS
  MOSTLY OF INPUT AND OUTPUT, PLUS A CALL TO THE ROUTINE AUTE
  WHICH FINDS THE SUBGROUP OF THE AUTOMORPHISM GROUP OF GRAPH Z
  STABILIZING THE EDGE FROM VERTEX 1 TO VERTEX 2 }

```

```

{*****}
*      Segment Sims      *
*      *                  *
*      This segment contains the routines which perform      *
*      Schreier's algorithm. A group is expanded as a tower of      *
*      vertex stabilizers.      *
*      *                  *
*      *****          *
*      {*****}          *
*      *                  *
*      FUNCTION MultPerms      *
*      *                  *
*      This function returns a pointer to the permutation product      *
*      of two permutations pointed at by h and g.      *
*      *                  *
*      NOTE: parameter n is the number of points being permuted.      *
*      *                  *
*      *****          *
*      FUNCTION MultPerms ( h: permutation; g: permutation;      *
*                          n: pt ): permutation;      *
*      *                  *
*      VAR      *
*      a: permutation;      *
*      i: pt;      *
*      *                  *
*      REF      *
*      ZSize: integer;      *
*      *                  *
*      BEGIN      *
*      a := ampxnew( ZSize*4 );      *
*      FOR i := 1 TO n DO      *
*          a[ i ] := g[ h[ i ] ];      *
*      MultPerms := a      *
*      END; { MultPerms }      *
*      *                  *
*      {*****}          *
*      *                  *
*      FUNCTION IGroup      *
*      *                  *
*      This function returns the identity group which is      *
*      represented by a based GROUP structure which has      *
*      - no generators (nil GENS pointer)      *
*      - stores Orbit of parameter p = < p >      *
*      and the coset representative of p is e      *
*      *                  *
*      NOTE: parameter n is the number of points being permuted.      *
*      *                  *
*      *****          *
*      *                  *
*      FUNCTION IGroup ( p: pt; n: pt ): Group;

```

```

VAR
  grp: Group;
  i: pt;
  ptrans: PtrTransElt;

REF
  ZSize: integer;

BEGIN
  new(grp);
  WITH grp@ DO
    BEGIN
      Degree := n;
      Order := 1;
      Gens := nil;
      Orbit := ampxnew( ZSize*4 );
      new( ptrans );
      ptrans@.a := ampxnew( ZSize*4 );
      ptrans@.ainverse := ampxnew( ZSize*4 );
      FOR i := 1 TO n DO
        BEGIN
          ptrans@.a[ i ] := i;
          ptrans@.ainverse[ i ] := i;
          Orbit[ i ] := nil;
        END; { FOR }
      Point := p;
      ptrans@.NextTransElt := 0;
      Trans.TransStart := p;
      Trans.TransEnd := p;
      Orbit[ p ] := ptrans;
      Stab := nil;
    END; { WITH }
  IGroup := grp;
END; { IGroup }

{*****}
*
*   PROCEDURE AddToGenerators
*
*   This function adds the generator G (a permutation)
*   to the generator list GENS. The inverse of the generator
*   is found.
*
*   {*****}

PROCEDURE AddToGenerators ( g: permutation;
  VAR Gens: PtrGenElt;
  n: pt);

VAR
  newgen: PtrGenElt;
  i: pt;

```

```

REF
  ZSize: integer;

BEGIN
  new ( newgen );
  newgen@.Gen := ampxnew( ZSize*4 );
  newgen@.GenInverse := ampxnew( ZSize*4 );
  FOR i := 1 TO n DO
    BEGIN
      newgen@.Gen[ i ] := g[ i ];
      newgen@.GenInverse[ g[ i ] ] := i;
    END; { FOR }
  IF Gens = nil THEN
    newgen@.GenSymbol := 'A'
  ELSE
    IF Gens@.GenSymbol = 'I' THEN
      newgen@.GenSymbol := 'J'
    ELSE
      IF Gens@.GenSymbol = 'R' THEN
        newgen@.GenSymbol := 'S'
      ELSE
        newgen@.GenSymbol := succ(Gens@.GenSymbol);
      newgen@.NextGenElt := Gens;
      Gens := newgen;
    END; { AddToGenerators }

{*****}
*
*   FUNCTION InvertPerm
*
*   This function returns the inverse of the given
*   permutation, g. The other parameter, n, is the number of
*   points being permuted.
*
*   {*****}

FUNCTION InvertPerm( g: permutation; n: pt ): permutation;

VAR
  gp: permutation;
  i: pt;

REF
  ZSize: integer;

BEGIN
  gp := ampxnew( ZSize*4 );
  FOR i := 1 TO n DO
    gp[ g[ i ] ] := i;
  InvertPerm := gp;
END; { InvertPerm }

{*****}
*

```

```

* FUNCTION Identity
*
* This function returns true if the permutation passed to it
* is the identity permutation, otherwise it returns false.
*
* NOTE: parameter n is the number of points being permuted.
*
*****}
FUNCTION Identity ( p: permutation; n: pt ): boolean;
VAR
  i: integer;
  identityperm: boolean;
BEGIN
  identityperm := ( p@[ 1 ] = 1 );
  i := 2;
  WHILE identityperm and ( i <= n ) DO
    BEGIN
      identityperm := ( p@[ i ] = i );
      i := i + 1;
    END; { WHILE }
  END; { WHILE }
  Identity := identityperm;
END; { Identity }

*****}
PROCEDURE CalculateOrder
*
* This procedure calculates the order of each group in the
* tower of stabilisers for G.
*
*****}
PROCEDURE CalculateOrder ( VAR G: Group );
VAR
  transize: integer;
  t: integer;
BEGIN
  IF G@.Gens <> nil THEN
    BEGIN
      { find transversal size }
      t := G@.Trans.TransStart;
      transize := 0;
      WHILE t <> 0 DO BEGIN
        transize := transize + 1;
        t := G@.Orbit[ t ]@.NextTransElt;
      END; { WHILE }
      IF G@.Stab = nil THEN
        G@.Order := transize
      ELSE
        BEGIN
          CalculateOrder( G@.Stab );
          G@.Order := transize * G@.Stab@.Order;
        END { IF }
      END; { CalculateOrder }
    END
  ELSE
    BEGIN
      CalculateOrder( G@.Stab );
      G@.Order := transize * G@.Stab@.Order;
    END { IF }
  END; { CalculateOrder }

*****}
PROCEDURE AddOrbit
*
* This procedure adds the point X to Orbit and the coset
* representative of X to the transversal. The coset represen-
* tative is the permutation H multiplied by the permutation G.
*
* NOTE: parameter n is the number of points being permuted.
*
*****}
PROCEDURE AddOrbit (x: pt;
  VAR Orbit: OrbitArray;
  VAR Trans: TransRec;
  h: permutation;
  g: permutation;
  n: pt );
VAR
  newTransElt: PtrTransElt;
BEGIN
  new(newTransElt);
  Orbit[ x ] := newTransElt;
  newTransElt@a := MultPerms( h, g, n );
  newTransElt@aInverse := InvertPerm( newTransElt@a, n );
  newTransElt@.NextTransElt := 0;
  Orbit[ Trans.TransEnd ]@.NextTransElt := x;
  Trans.TransEnd := x;
END; { AddOrbit }

*****}
FUNCTION GroupElement
*
* This function returns true if perm G is an element of
* group GRP, otherwise it returns false.
*
* ALGORITHM
*
* VAR
*   H: group;
*   h: permutation;
*   ginGRP, done: boolean;
*
* BEGIN

```

```

ginGRP, done := false;
h := g;
H := GRP;
WHILE NOT done DO
  BEGIN
    IF h = identity THEN BEGIN
      ginGRP := true; < the identity is in all groups >
      done := true
    END < IF >
    ELSE BEGIN
      WITH H DO BEGIN
        k := Point;
        x := (k)h;
        IF x is not in Orb(k) THEN done := true
        ELSE BEGIN
          a := coset rep corresponding to x;
          h := h a;
          H := Stab(k);
        END < ELSE >
      END < WITH >
    END < ELSE >
  END < WHILE >
RETURN( ginGRP );
END; < GroupElement >
*****]

```

```

FUNCTION GroupElement ( g: permutation;
  grp: Group): boolean;

```

```

VAR
  hgrp: Group;
  h, hp: permutation;
  gingrp, done: boolean;
  x: pt;

```

```

REF
  ZSize: integer;

```

```

BEGIN
  gingrp := false;
  done := false;
  h := ampnew( ZSize*4 );
  FOR x := 1 to ZSize DO
    h[x] := g[x];
  hgrp := grp;
  WHILE not done DO
    BEGIN
      IF Identity( h, hgrp@.Degree ) THEN
        BEGIN { identity is in all groups }
          gingrp := true;
          done := true;
        END { IF }
      ELSE
        WITH hgrp@ DO
          BEGIN
            x := h@.Point;
            IF Orbit[ x ] = nil THEN
              done := true { x is not in Orb(Point) }
            ELSE
              BEGIN { x is in Orb(Point) }
                hp := MultPerms( h, Orbit[ x ]@.aInverse,
                  hgrp@.Degree );
                dispose( h );
                h := hp;
                hgrp := hgrp@.Stab;
              END { ELSE }
            END { WITH }
          ELSE
            done := true
          END; { WHILE }
        dispose( h );
        GroupElement := gingrp;
      END; { GroupElement }
    END
  END

```

```

ELSE { h is not the identity }
  IF hgrp <> nil THEN
    IF hgrp@.Gens = nil THEN
      done := true
    ELSE
      WITH hgrp@ DO
        BEGIN
          x := h@.Point;
          IF Orbit[ x ] = nil THEN
            done := true { x is not in Orb(Point) }
          ELSE
            BEGIN { x is in Orb(Point) }
              hp := MultPerms( h, Orbit[ x ]@.aInverse,
                hgrp@.Degree );
              dispose( h );
              h := hp;
              hgrp := hgrp@.Stab;
            END { ELSE }
          END { WITH }
        ELSE
          done := true
        END; { WHILE }
      dispose( h );
      GroupElement := gingrp;
    END; { GroupElement }
  END

```

```

*****
PROCEDURE AddGen

```

```

This procedure expands the group GRP by adding a new
generator, G. Each new chord the new generator creates in
the Schreier coset diagram for STAB(POINT) (same as the
Orbit of POINT) with the tree given by the coset represen-
tatives creates another generator for STAB(POINT) (and is
added to STAB(POINT) by a recursive call to ADDGEN as long
as it is not a redundant generator).
The Orbit of POINT is expanded by applying the new
generator to each element already in the Orbit. If the
new generator merely takes a point in the Orbit to another
point in the Orbit then a new chord has been found.

```

```

VARIABLE DICTIONARY

```

```

g - parameter - a generator in permutation form
grp - parameter - the group to which g should be added

gp - a generator for the stabiliser of group grp found while
      expanding the Orbit in grp using grp's new generator g
      - in permutation form
a - pointer to group structure used to apply each generator
      of grp to new Orbit elements

```

```

ALGORITHM

```

```

* AddGen(g: permutation; G: group);
*
* < adds generator g to group G >
*
* BEGIN
*   WITH G DO BEGIN
*     add g to Gens, the generators of G;
*     FOR each k in Orbit DO BEGIN
*       x := (k)g;
*       IF x is not in Orbit THEN BEGIN
*         add x to Orbit;
*         construct coset representative for x
*       END < IF >
*     ELSE BEGIN
*       compute g', a new generator for Stab(Point)
*       IF NOT GroupElement(g', Stab) THEN
*         AddGen(g', Stab)
*       END < ELSE >
*     END; < FOR >
*   IF generator g has extended Orbit THEN
*     FOR each new point k in Orbit DO
*       FOR each a in Gens DO BEGIN
*         x := (k)a;
*         IF x is not in Orbit THEN BEGIN
*           add x to Orbit;
*           construct coset representative for x
*         END < IF >
*       ELSE BEGIN
*         compute g', a new generator for Stab(Point)
*         IF NOT GroupElement(g', Stab) THEN
*           AddGen(g', Stab)
*         END < ELSE >
*       END < FOR >
*     END < WITH >
*   END; < AddGen >
*
* *****]
*
* PROCEDURE AddGen( g: permutation;
*   VAR grp: Group;
*   base: BaseArray;
*   VAR BaseIndex: integer);
*
* VAR
*   x: pt;
*   q: integer;
*   gp: permutation;
*   b: PtrGenElt;
*   qstart: integer;
*   p: permutation;
*
* REF
*   ZSize: integer;

```

Output: text;

```

BEGIN
  IF grp = nil THEN BEGIN
    grp := IGroup( base@[ BaseIndex ], ZSize );
    BaseIndex := BaseIndex + 1
  END; { IF }
  qstart := 0;
  WITH grp@ DO
    BEGIN
      AddToGenerators(g, Gens, grp@.Degree );
      q := Trans.TransStart;
      WHILE q <> qstart DO
        BEGIN
          x := g@[ q ];
          IF Orbit@[ x ] = nil THEN { x is not in Orbit }
            BEGIN
              AddOrbit( x, Orbit, Trans, Orbit@[ q ]@.a, g,
                grp@.Degree );
              IF qstart = 0 THEN
                qstart := x
              END { IF }
            ELSE { got a chord in Orbit from q to x
              where the tree is defined by the coset
              reps corresponding to each element in
              Orbit.
            }
            BEGIN
              p := MultPerms( g, Orbit@[ x ]@.aInverse,
                grp@.Degree );
              gp := MultPerms( Orbit@[ q ]@.a, p, grp@.Degree );
              dispose( p );
              IF Stab = nil THEN BEGIN
                Stab := IGroup( base@[ BaseIndex ],
                  grp@.Degree );
                BaseIndex := BaseIndex + 1
              END; { IF }
              IF not GroupElement( gp, Stab ) THEN
                AddGen( gp, Stab, base, BaseIndex );
                dispose( gp );
              END; { ELSE }
              q := Orbit@[ q ]@.NextTransElt
            END; { WHILE }
          WHILE qstart <> 0 DO
            BEGIN
              b := Gens;
              WHILE b <> nil DO
                BEGIN
                  x := b@.Gen@[ qstart ];
                  IF Orbit@[ x ] = nil THEN { x is not in Orbit }
                    BEGIN
                      AddOrbit( x, Orbit, Trans, Orbit@[ qstart ]@.a, b@.Gen,
                        grp@.Degree );
                    END { IF }
                  ELSE

```



```

BEGIN
  p := MultPerms( b@.Gen, Orbit@[ x ]@.ainverse,
    grp@.Degree );
  gp := MultPerms( Orbit@[ qstart ]@.a, p, grp@.Degree );
  dispose( p );
  IF Stab = nil THEN BEGIN
    Stab := IGroup( base@[ BaseIndex ],
      grp@.Degree );
    BaseIndex := BaseIndex + 1;
    END; { IF }
    IF not GroupElement( gp, Stab ) THEN
      AddGen( gp, Stab, base, BaseIndex );
      dispose( gp );
    END; { ELSE }
    b := b@.NextGenElt;
    END; { WHILE }
    qstart := Orbit@[ qstart ]@.NextTransElt;
    END; { WHILE }
  END { WITH }
  AddGen;
END;

*****
FUNCTION GroupGen
*****
  This function is passed a list of (possible) generators
  for a group. Some of the generators may be redundant.
  This function returns the group generated by the generators
  but with no redundant generators.
  The algorithm used is Schreier's algorithm, which builds
  up the group generator by generator, as a tower of
  stabilisers.
  This function is also passed a base which tells the
  algorithm which stabilisers are to be used in the tower and
  what order they should be put into the tower.
  NOTE: parameter n is the number of points being permuted.
  ALGORITHM:
  grp := identity group;
  FOR each g in GenList DO
    IF g is not in grp THEN
      < add the generator to grp >
      AddGen(g, grp)
  *****
FUNCTION GroupGen( GenList: PtrGenElt; n: pt; base: BaseArray ):
  Group;
VAR
  grp: Group;
  g: PtrGenElt;
  BaseIndex: integer;

```

```

REF
  ZSize: integer;
  Output: text;
BEGIN
  grp := IGroup( base@[ 1 ], n );
  BaseIndex := 2;
  g := GenList;
  WHILE g <> nil DO
    BEGIN
      IF not GroupElement( g@.Gen, grp ) THEN
        AddGen( g@.Gen, grp, base, BaseIndex );
        g := g@.NextGenElt;
      END; { WHILE }
    CalculateOrder( grp );
    GroupGen := grp;
  END; { GroupGen }
  *****
  FUNCTION Aute
  *****
  This function is passed a graph Z and the number of vertices
  in graph Z (parameter ZSize). This function finds the subgroup
  of the automorphism group of Z which fixes the edge e between
  vertices 1 and 2, and returns the group as a tower of point
  stabilizers.
  The vertices are partitioned according to the distance from
  an imaginary vertex v0 on the edge joining vertices 1 and 2.
  v1 represents all vertices at distance i from v0.
  A "tower" of subgraphs of Z is created as follows:
  Xk is the subgraph of Z with vertex set v1+v2+...+vk and edge
  set E[Z[v1+v2+...+vk-1]] and all edges in Z from vk-1 to vk,
  written [vk-1, vk].
  The identity subgroup is expanded to be Aute of X1, then
  expanded again to be Aute of X2, ..., until Aute of Z is found.
  Let Gk be the subgroup of the automorphism group of Xk fixing
  edge e. Let Fk: Gk+1 -> Gk be the homomorphism restricting
  elements of Gk+1 (automorphisms of graph Xk+1) to graph Xk.
  The expansion of group Gk to group Gk+1 is done by finding the
  kernel of the homomorphism Fk and by finding the image of Fk
  (which is expanded to elements of Gk+1/Ker(Fk)).
  Ker(Fk) is generated by transpositions consisting of two
  vertices in vk+1 which are connected to exactly the same
  vertices in vk. Im(Fk) is the subgroup of Gk which set-wise
  stabilizes various sets in vk, vk, and vk.
  The sets represent the edges in graph Xk+1 which are not a part
  of graph Xk. Thus, the set of all edges between two vertices in
  vk in Xk+1 must be stabilized in vk. The set of all vertices
  x in vk which are joined to vertices y in vk+1 where y is joined
  only to x in vk must be stabilized in vk. The set of all pairs

```

```

* of vertices (x,y) in  $V_k$  which are both joined to a vertex u in  $V_{k+1}$  where u is joined only to x and y in  $V_k$  must be stabilized.
* The set of all triples of vertices ( x, y, z ) in  $V_k$  which are
*   (3)
* connected to a vertex u in  $V_{k+1}$  must be stabilized in  $V_k$ .
* These sets can be broken down still more by considering the
* orbits of vertices in  $V_k$  caused by  $G_k$ . For instance, an edge in
*  $V_{k+1}$  which joins vertices x and y, where x and y are both in
*  $V_k$  and x is in orbit  $O_i$  and y is in orbit  $O_j$ , can only be mapped
* to another edge  $e=uv$  where u is in  $O_i$  and v is in  $O_j$ .
* Once all the sets are stabilized,  $im(F_k)$  has been found and
* can be extended to  $G_{k+1}/Ker(F_k)$ .  $G_{k+1}/Ker(F_k)$  and  $Ker(F_k)$ 
* give us  $G_{k+1}$ .
*
* variable dictionary:
* V1, V2: represent partitions  $V_i$  and  $V_{i+1}$ 
*   The records in Z representing the vertices in  $V_i$  are
*   moved so that they are contiguous. Thus, a partition
*   is represented by the indices in Z of the first and
*   last vertices in the partition.
* IndexMap: a mapping from a vertex to its index in graph Z.
* VertexMap: A mapping from a vertex in  $V_1$  to its merge-find
*   pointer in the structure representing the orbits of
*    $G_k$  in  $V_1$ .
* Ker: a linked list of generators for  $Ker(F_k)$ .
* Sets: used to organize the sets of points/edges/triples
*   according to type and orbits as the sets are being formed.
* SchType, SchStabType, StabType: used to reorganize the sets
*   according to the amount of work required to stabilize each
*   set.
*
* ALGORITHM:
*
* G := < ( 1, 2 ) >;      ( group  $G_1$  )
* FOR each graph  $X_k$  DO BEGIN
*   - find  $Ker(F_k)$  and the sets which must be stabilized
*   - to find  $im(F_k)$ 
*   - reorganize the sets according to the amount of work to
*     stabilize each one.
*   - choose the set requiring the least amount of work
*   WHILE there is a set to be stabilized DO BEGIN
*     G := the subgroup of G stabilizing the chosen set
*     find the next best set
*     find the orbits in  $V_k$  of the new group G
*     reorganize the best set according to the new orbits (this
*       will probably split the set into smaller sets, take
*       the best subset)
*   END; ( WHILE )
*   expand the elements of G to map the vertices of  $V_{k+1}$ 
*   G := <G,  $Ker(F_k)$ >
* END; ( FOR )
* *****

```

```

FUNCTION AutE( Z: graph; ZSize: integer ): Group;

```

```

CONST
  HugeSize = 10000;

TYPE
  { PARTITIONING THE VERTICES ACCORDING TO DISTANCE FROM THE
    CHOSEN EDGE
  }
  VTypePtr = @VType;
  VType = RECORD
    vstart, vend: integer
    { info about 1 such vertex
      partition
      index in graph array of
      1st + last vertex in this
      partition
    }
  END;

  { ORBITS OF VERTICES WITHIN A PARTITION }
  OrbitTypePtr = @OrbitType;
  MForbitPtr = @MForbitType;
  OrbitEltPtr = @OrbitEltType;
  OrbitType = array[ 1..hugeSize ] of MForbitPtr;
  { used to map vertex to the
    merge-find elt representing
    it
  }
  MForbitType = RECORD
    CASE TopVertex: boolean OF
      false: (ParentVertex: MForbitPtr);
      true: (OrbitSize: integer;
             OrbitList: OrbitEltPtr)
    { ptr to a circular doubly
      linked list of vertices
      in the orbit
    }
  END;
  OrbitEltType = RECORD
    OVertex: integer;
    OLink, OBLink: OrbitEltPtr
  END;
  OArrayTypePtr = @OArrayType;
  OArrayType = array[ 1..hugeSize ] of integer;

  { BASE TO PASS TO SCHREIER'S ALGORITHM TO TELL IT WHICH
    STABILISERS TO CREATE IN WHICH ORDER
  }
  BaseArray = @BaseArrayType;
  BaseArrayType = array[ 1..hugeSize ] of integer;

  { SETS OF POINTS/EDGES/TRIPLES (NTUPLES) TO BE STABILIZED }
  ntuple = array[ 1..3 ] of integer;

```

```

ntupleType = ( Pink,      { a point at distance i from
                          the chosen edge connected to
                          only one point at distance i+1 }
  Brown1,    { an edge between two points at
               distance i (original edge) in
               which the two points are in the
               same orbit }
  Brown2,    { similar to Brown1 except that
               the two points are in two
               different orbits }
  GreenBrown1, { similar to Brown1 except that
                 the two points are connected
                 to a vertex at distance i+1
                 and that vertex is connected
                 only to the two points at
                 distance i }
  GreenBrown2, { similar to Brown2 except that
                 the two points are also
                 connected to a point at distance
                 i+1 which is connected to no
                 other points at distance i
                 other than the two given }
  Green1,    { two points at distance i from
               the fixed edge which are both
               connected to a point at
               distance i+1 (and it is connected
               to no points at distance i
               other than the two given).
               The two points are in the same
               orbit. }
  Green2,    { similar to Green1 except that
               the two points are in different
               orbits }
  Red1,      { three points at distance i from
               the fixed edge connected to a
               point at distance i+1 (and is
               connected to no points at
               distance i other than the 3
               given points). The 3 points
               are all in the same orbit }
  Red2,      { similar to Red1 except that
               2 of the points are in 1 orbit
               and the 3rd is in a different
               orbit }
  Red3);     { similar to Red1 except each of
               the 3 points is in a different
               orbit than the other 2 }

ntupleSetPtr = @ntupleSet;
SetEltPtr = @SetEltType;
ntupleSet = RECORD
  single: boolean; { a set of ntuples to stabilize }
                  { Brown edges are always single
                    a Pink, Green, or Red ntuple
                    is double if all points in the

```

```

                          ntuple are connected to the
                          same TWO points at distance
                          i+1 rather than just one. }
  setkind: ntupleType;
  setsize: integer; { how many ntuples to stabilize? }
  typetotal: integer;
                  { how many ntuples of that
                    particular type? }
  setlist: seteltPtr;
                  { ptr to a linked list of ntuples }
  ntuplesize: integer;
                  { how many points in the ntuples }
  nextset: ntupleSetPtr
END;

SetEltType = RECORD
  SetElt: ntuple;
  NextSetElt: SetEltPtr
END;

SetTypePtr = @SetType;
SetType = array[ 1..HugeSize ] of ntuple;
                  { set of ntuples to be passed to
                    the set stabilizing routines }

InitSet = array[ ntupleType ] of ntupleSetPtr;
                  { used to organize the sets as
                    they are being created }

Mftuple = array[ 1..3 ] of MForbitPtr;
                  { pointers to the tops of merge-
                    find trees of the vertices in
                    an ntuple }

{ A MAPPING OF VERTICES TO THE INDICES OF THEIR RECORDS IN
  THE GRAPH STRUCTURE Z }

IndexMapPtr = @IndexMapType;
IndexMapType = array[ 1..HugeSize ] of integer;

{ BIT ARRAY USED WHEN EXTENDING GENERATORS OF IM(Fk) TO
  GENERATORS OF Gk+1/KER(Fk) }

MapType = array[ 1..hugeSize ] of boolean;
MapTypePtr = @MapType;
                  { USED TO KNOW WHICH POINTS THE
                    PERMUTATIONS IN G HAVE ALREADY
                    BEEN EXTENDED TO MAP }

```

```

VAR
  F, G: PtrGenElt;
  H: Group;
  p: permutation;

```

```

i,j,k,l: integer;
base: BaseArray;
BaseIndex: integer;
Ker: PtrGenElt;

{ a base to pass to Schreier's Alg }
{ current posn in base }
{ the kernel of the mapping from }
{ the group which is AutE of the }
{ graph X consisting of }
{ i+1 }
{ vertices of graph Z }
{ at distance i+1 or less from the }
{ fixed edge and all induced edges }
{ of Z except the ones between }
{ vertices at distance i+1 }
{ to the group which is AutE of the }
{ graph X }
{ }

Sets: InitSet; { organize the sets as they're }
                { being formed, partitioning them }
                { on the ntuple type }
SchType, SchStabType, StabType: ntupleSetPtr;
                { after the sets have all been }
                { created, organize them according }
                { to the amount of work required to }
                { stabilize them: }
                { 1) sets consisting of one ntuple }
                {   of type Pink, Brown2, Green2, }
                {   Red3, or GreenBrown2 }
                {   can be stabilized by a call to }
                {   Schreier's alg. }
                { 2) sets consisting of one ntuple }
                {   of type Red2 requires one call }
                {   to Schreier's alg and one call }
                {   to the set stabilizing routine. }
                { 3) all other sets require calls to }
                {   the set stabilizing routine. }
                { }

s: PtrGenElt;

a, aIndex, aMap, aMapIndex: ntuple;
aSize: integer;
q: PtrGenElt;
Mfa: MFTuple;
se: SetEltPtr;

T: ntupleSetPtr;
Mapped: MapTypePtr;

DEF
VertexMap: OrbitTypePtr; { map a vertex to its MF ptr }
                          { used to find which orbit it's in }
IndexMap: IndexMapPtr; { map vertex to index in Z }
V1, V2: VTypePtr;

{ SOME OF THE PROCEDURES AND FUNCTIONS CALLED BY FUNCTION AUTE }
{ HAVE NOT BEEN INCLUDED: }

```

ROUTINES WHICH ORDER THE SETS OF NTUPLES ACCORDING TO THE
AMOUNT OF WORK REQUIRED TO STABILIZE THE SETS.

OrderSets: This procedure is passed the newly found sets
of ntuples sorted according to ntuple type and the number
of orbits used (parameter Sets). It sorts the sets
according to the amount of work required to stabilize
each set (i.e. sets up the lists of sets SchType, SchStabType,
and StabType).

ReOrderBestSet: This procedure chooses the best set (one requiring
the least amount of work to stabilize) and reorganizes it
according to the new orbits containing the vertices of the
ntuples.

ROUTINES WHICH ARE PASSED A SET WHICH IS TO BE STABILIZED AND
SET UP THE PARAMETERS FOR A CALL TO THE APPROPRIATE STABILIZING
ROUTINE (SCHREIER'S ALGORITHM OR THE SETWISE STABILIZING ROUTINES).

StabSch: sets up the calls for sets which can be stabilized using
Schreier's algorithm.

StabSchSt: sets up the calls for sets which contain one triple
whose points are contained in two orbits (requires one call to
Schreier's algorithm and another to the SetStab routines).

StabPink: sets up the calls to the SetStab routines for sets
of more than one 1-tuples.

StabE1: sets up the calls for sets containing 2-tuples whose
vertices are contained in only one orbit.

StabE2: sets up the calls to the SetStab routines for sets
containing 2-tuples whose vertices are in two orbits.

StabRed1, StabRed2, StabRed3: set up the calls to the SetStab
routines for sets containing triples whose vertices are
contained in 1, 2 or 3 orbits, respectively. (In the case
of StabRed2, the set contains more than one triple. If the
set contains only one triple, it is handled by routine
StabSchSt.)

ROUTINES WHICH DISPOSE OF STRUCTURES WHICH ARE NO LONGER NEEDED

DispSet: dispose of a structure representing a set (TYPE
ntupleSetPtr).

DisposeMForbits: dispose of the merge/find structure used to
find the orbits of the vertices in partition V1.

DisposeTower: dispose of a group in the form of a tower of
point stabilizers.

DispGenList: dispose of a group given as a linked list of
generators.

```

{*****}
*
*  MAIN LINE OF AUTE
*
{*****}

```

BEGIN

```

{ CREATE THE MAPPING FROM A VERTEX TO ITS INDEX IN THE GRAPH Z }
IndexMap := ampxnew( ZSize*4 );
FOR i := 1 TO ZSize DO
  IndexMap@[ i ] := i;
{ CREATE THE FIRST PARTITION V1 CONTAINING VERTICES 1 AND 2 }

new( V1 );
V1@.VStart := 1;
V1@.VEnd := 2;

{ CREATE GROUP G WHICH IS AUTE OF THE GRAPH X CONSISTING OF
  THE VERTICES 1 AND 2. }

new( G );
G@.Gen := ampxnew( ZSize*4 );
G@.GenInverse := ampxnew( ZSize*4 );
FOR i := 3 TO ZSize DO BEGIN
  G@.Gen@[ i ] := i;
  G@.GenInverse@[ i ] := i;
END;
{ FOR }
G@.Gen@[ 1 ] := 2;
G@.GenInverse@[ 1 ] := 2;
G@.Gen@[ 2 ] := 1;
G@.GenInverse@[ 2 ] := 1;
G@.NextGenElt := nil;

{ CREATE A BASE WHICH CAN BE USED BY SCHREIER'S
  ALGORITHM WHEN REDUCING THE NUMBER OF GENERATORS
  REQUIRED TO REPRESENT GROUP Gk+1 }

base := ampxnew( ZSize*4 );
baseIndex := 1;
FOR i := 1 TO ZSize DO
  base@[ i ] := i;

{ CREATE A BIT ARRAY WHICH WILL TELL US WHICH VERTICES
  WE HAVE ALREADY MAPPED WHEN EXTENDING GENERATORS OF
  IM(Fk) TO GENERATORS OF Gk+1/KER(Fk) }

Mapped := ampxnew( ZSize );
FOR i := 2 TO ZSize DO Mapped@[ i ] := false;
Mapped@[ 1 ] := true;
Mapped@[ 2 ] := true;

{ WORK WITH EACH PARTITION IN TURN, FINDING AUTE OF EACH GRAPH
  X IN TURN UNTIL AUTE OF THE ENTIRE GRAPH Z IS FOUND }

```

WHILE V1@.VEnd <> ZSize DO BEGIN

```

{ FIND THE ORBITS OF THE VERTICES IN V1 }

CreateVertexMap( V1, VertexMap );
FOrbits( V1, VertexMap, G );

{ FIND THE NEXT PARTITION, V2, AND AT THE SAME TIME FIND
  THE KERNEL OF THE MAPPING FROM THE NEXT GROUP UP TO THE
  CURRENT GROUP, AND THE SETS TO BE STABILIZED TO FIND G/KER }

FindNextPartition( V1, V2, VertexMap, Ker, Sets );

{ REORDER THE NEWLY FOUND SETS SO THE SET WHICH REQUIRES
  A LESSER AMOUNT OF PROCESSING TO STABILIZE CAN BE FOUND }

SchType := nil;
SchStabType := nil;
StabType := nil;
OrderSets( Sets, SchType, SchStabType, StabType );

{ STABILIZE EACH OF THE SETS }

WHILE ((SchType <> nil) or (SchStabType <> nil) or
  (StabType <> nil)) and (G <> nil) DO BEGIN
  IF SchType <> nil THEN BEGIN
    StabSch( SchType, G );
    T := SchType;
    SchType := SchType@.NextSet;
  END ELSE IF SchStabType <> nil THEN BEGIN
    StabSchSt( SchStabType, VertexMap, V1, G );
    T := SchStabType;
    SchStabType := SchStabType@.NextSet;
  END ELSE BEGIN
    CASE StabType@.SetKind OF
      Pink: StabPink( StabType, VertexMap, V1, G );
      Brown1, GreenBrown1, Green1:
        StabE1( StabType, VertexMap, V1, G );
      Brown2, GreenBrown2, Green2:
        StabE2( StabType, VertexMap, V1, G );
      Red1: StabRed1( StabType, VertexMap, V1, G );
      Red2: StabRed2( StabType, VertexMap, V1, G );
      Red3: StabRed3( StabType, VertexMap, V1, G );
    END; { CASE }
    T := StabType;
    StabType := StabType@.NextSet;
  END; { IF }
  DispSet( T );

  { FIND THE NEW ORBITS }

  ReOrderMForbits( VertexMap, V1 );
  FOrbits( V1, VertexMap, G );

```

```

{ REORGANIZE THE NEXT SET NOW THAT THE ORBITS HAVE CHANGED }
IF (SchType <> nil) or (SchStabType <> nil) or
  (StabType <> nil) THEN BEGIN
  ReOrderBestSet( SchType, SchStabType, StabType, Sets,
    V1, VertexMap );
  OrderSets( Sets, SchType, SchStabType, StabType );
END; { IF }
END; { WHILE }
WHILE SchType <> nil DO BEGIN
  T := SchType;
  SchType := SchType@.NextSet;
  DispSet( T )
END;
WHILE SchStabType <> nil DO BEGIN
  T := SchStabType;
  SchStabType := SchStabType@.NextSet;
  DispSet( T )
END;
WHILE StabType <> nil DO BEGIN
  T := StabType;
  StabType := StabType@.NextSet;
  DispSet( T )
END; { WHILE }
DisposeMForbits( VertexMap, V1 );
{ EXTEND EACH PERMUTATION IN G TO MAP V2 }
IF G <> nil THEN
  FOR i := V2@.VStart to V2@.VEnd DO
    IF not Mapped@[ Z@[ i ]@.Vertex ] THEN BEGIN
      FindAdj( i, a, aSize, V1 );
      FOR j := 1 to aSize DO BEGIN
        aIndex[ j ] := IndexMap@[ a[ j ] ];
      END;
      j := FindMatch( a, aIndex, aSize, V1, V1@.VEnd,
        Z@[ i ]@.Vertex );
      Mapped@[ Z@[ i ]@.Vertex ] := true;
      IF j <> 0 THEN BEGIN
        Mapped@[ j ] := true;
      END; { IF }
      q := G;
      WHILE q <> nil DO BEGIN
        FOR k := 1 to aSize DO BEGIN
          aMap[ k ] := q@.Gen@[ a[ k ] ];
          aMapIndex[ k ] := IndexMap@[ aMap[ k ] ];
        END; { FOR }
        k := FindMatch( aMap, aMapIndex, aSize, V1, V1@.VEnd,
          0 );
        q@.Gen@[ Z@[ i ]@.Vertex ] := k;
        q@.GenInverse@[ k ] := Z@[ i ]@.Vertex;
        IF j <> 0 THEN BEGIN
          l := FindMatch( aMap, aMapIndex, aSize, V1,
            V1@.VEnd, k );
          q@.Gen@[ j ] := l;
          q@.GenInverse@[ l ] := j;
        END; { IF }
        q := q@.NextGenElt;
      END; { WHILE }
    END; { IF }
  END;
  { ADD GENERATORS OF KER(Fk) TO GENERATORS OF
    Gk+1/KER(Fk) TO GET GENERATORS FOR Gk+1 }
  baseIndex := 2;
  H := IGroup( base@[ 1 ], ZSize );
  WHILE G <> nil DO BEGIN
    F := G@.NextGenElt;
    IF not GroupElement( G@.Gen, H ) THEN BEGIN
      AddGen( G@.Gen, H, base, baseIndex );
    END; { IF }
    Dispose( G@.Gen );
    Dispose( G@.GenInverse );
    Dispose( G );
    G := F;
  END; { WHILE }
  WHILE Ker <> nil DO BEGIN
    IF not GroupElement( Ker@.Gen, H ) THEN
      AddGen( Ker@.Gen, H, base, baseIndex );
    s := Ker@.NextGenElt;
    Dispose( Ker@.Gen );
    Dispose( Ker@.GenInverse );
    Dispose( Ker );
    Ker := s;
  END; { WHILE }
  G := H@.Gens;
  H@.Gens := nil;
  DisposeTower( H );
  dispose( V1 );
  V1 := V2
END; { WHILE }
IF G <> nil THEN BEGIN
  { STABILIZE SETS OF EDGES WHOSE ENDPOINTS ARE BOTH IN THE
    FINAL PARTITION }
  CreateVertexMap( V1, VertexMap );
  Sets[ Pink ] := nil;
  Sets[ Brown1 ] := nil;
  Sets[ Brown2 ] := nil;
  Sets[ GreenBrown1 ] := nil;
  Sets[ GreenBrown2 ] := nil;
  Sets[ Green1 ] := nil;
  Sets[ Green2 ] := nil;
  Sets[ Red1 ] := nil;
  Sets[ Red2 ] := nil;
  Sets[ Red3 ] := nil;
  MFa[ 3 ] := nil;

```

```

FORbits( V1, VertexMap, G );
FOR i := V1@VStart to (V1@VEnd - 1) DO BEGIN
  FOR j := 1 to 3 DO BEGIN
    MFA[ 1 ] := Forbit( i-V1@VStart+1, VertexMap );
    IF (k >= V1@VStart) and (k > i) THEN BEGIN
      MFA[ 2 ] := Forbit( k-V1@VStart+1, VertexMap );
      new( se );
      se@.SetElt[ 1 ] := Z@[ i ]@.Vertex;
      se@.SetElt[ 2 ] := Z@[ k ]@.Vertex;
      IF MFA[ 1 ] = MFA[ 2 ] THEN
        AddToSets( se, MFA, Brown1, true, 2, Sets,
          VertexMap, V1, IndexMap );
      ELSE
        AddToSets( se, MFA, Brown2, true, 2, Sets,
          VertexMap, V1, IndexMap );
    END; { IF }
  END; { FOR }
END; { FOR }
SchType := nil; SchStabType := nil; StabType := nil;
OrderSets( Sets, SchType, SchStabType, StabType );
WHILE ((SchType <> nil) or (StabType <> nil)) and
  (G <> nil) DO BEGIN
  IF SchType <> nil THEN BEGIN
    StabSch( SchType, G );
    T := SchType;
    SchType := SchType@.NextSet;
  END ELSE BEGIN
    IF StabType@.SetKind = Brown1 THEN
      StabE1( StabType, VertexMap, V1, G )
    ELSE
      StabE2( StabType, VertexMap, V1, G );
    T := StabType;
    StabType := StabType@.NextSet;
  END; { ELSE }
  DispSet( T );
  ReOrderMForbits( VertexMap, V1 );
  FORbits( V1, VertexMap, G );
  IF ((SchType <> nil) or (StabType <> nil)) and
    (G <> nil) THEN BEGIN
    ReOrderBestSet( SchType, SchStabType, StabType,
      Sets, V1, VertexMap );
    OrderSets( Sets, SchType, SchStabType, StabType );
  END; { IF }
END; { WHILE }
WHILE (SchType@.NextSet;
  T := SchType@.NextSet;
  DispSet( SchType );
  SchType := T
END;
WHILE StabType <> nil DO BEGIN
  T := StabType;
  StabType := StabType@.NextSet;
  DispSet( T );
END;

END;
DisposeMForbits( VertexMap, V1 );
END; { IF }
AutoE := GroupGen( G, ZSize, base );
END; { AutoE }

[*****]
* Segment fnp (Find Next Partition)
*
* This segment contains the routines which find the next
* partition (V2) of vertices which are at distance i+1 from the
* vertex v0. The sets of ntuples which must be stabilized are
* also found and the kernel of the homomorphism Fk.
* [*****]

SEGMENT fnp;
[*****]
* FUNCTION FindMatch
*
* a is an ntuple of points in partition V1 (n = aSize). p is a
* vertex connected only to the ntuple a in V1. Find and return
* another vertex q that is also connected only to the ntuple a in
* V1 and also such that the index of q in Z is greater than
* EndIndex. If there is no such q then return 0.
* [*****]

FUNCTION FindMatch( a: ntuple; aIndex: ntuple; aSize: integer;
  V1: VTypePtr; EndIndex: integer; p: integer):
  integer;

VAR
  q: integer;
  i,j,k: integer;
  Matcha: array[ 1..3 ] of boolean;
  OtherVertex: boolean;
  qIndex: integer;
  b,bIndex: integer;

REF
  Z: Graph;
  ZSize: integer;
  IndexMap: IndexMapPtr;

BEGIN

```



```

{ CHECK EACH VERTEX ADJACENT TO THE 1ST VERTEX IN NTUPLE A
  TO SEE IF IT IS THE REQUIRED Q }

q := 0;
i := 1;
WHILE (q = 0) and (i <= 3) DO BEGIN
  q := Z@[ aindex[ 1 ] ]@.adjVertex[ i ];
  qindex := IndexMap@[ q ];
  IF (qindex > Endindex) and (q <> p) THEN BEGIN
    FOR j := 1 to aSize DO
      Matcha[ j ] := false;
      OtherVertex := false;
      j := 1;
      WHILE (j <= 3) and (not OtherVertex) DO BEGIN
        b := Z@[ qindex ]@.AdjVertex[ j ];
        bindex := IndexMap@[ b ];
        IF (bindex >= V1@.VStart) and (bindex <= V1@.VEnd)
          THEN BEGIN
            k := 1;
            OtherVertex := true;
            WHILE (k <= aSize) and OtherVertex DO
              IF a[ k ] = b THEN BEGIN
                OtherVertex := false;
                Matcha[ k ] := true;
                END { IF }
              ELSE
                k := k + 1;
            END; { IF }
            j := j + 1;
          END; { WHILE }
        IF OtherVertex THEN BEGIN
          q := 0;
          i := i + 1;
        END ELSE
          FOR j := 1 to aSize DO BEGIN
            IF not Matcha[ j ] THEN BEGIN
              q := 0;
              i := i + 1;
            END { IF }
          END { FOR }
        END ELSE BEGIN
          q := 0;
          i := i + 1;
        END { ELSE }
      END; { WHILE }
      FindMatch := q;
    END; { FindMatch }
  }
  *****
  *
  * PROCEDURE AddToSets
  *
  * Find a set of ntuples of the same type as ntuple a and using
  *
  *****

```

```

* the same orbits as ntuple a (pointed at by MFa) or create a new
* one. Then add ntuple a to it.
*
*****

```

```

PROCEDURE AddToSets( a: SetEltPtr; MFa: MFTuple; aSetType: ntupleType;
  aSingle: boolean; aSize: integer;
  VAR Sets: InitSet; VertexMap: OrbitTypePtr;
  V1: VTypePtr; IndexMap: IndexMapPtr );

```

```

VAR
  CorrectSet: ntupleSetPtr;
  found: boolean;
  MatchOrbit: array[ 1..3 ] of boolean;
  i,j,k: integer;
  newMF: MFOrbitPtr;
  OtherOrbit: boolean;

```

```

REF
  Z: Graph;
  ZSize: integer;

```

```

BEGIN
  {SEE IF A SET OF THE SAME TYPE ALREADY EXISTS }

  CorrectSet := Sets[ aSetType ];
  found := false;
  WHILE (not found) and (CorrectSet <> nil) DO BEGIN
    IF (CorrectSet@.single and aSingle) or
      ((not CorrectSet@.Single) and (not aSingle)) THEN BEGIN
      found := true;
      FOR j := 1 to aSize DO
        MatchOrbit[ j ] := false;
      { MAKE SURE THAT THE ORBITS OF THE SET BEING EXAMINED MATCH
        THOSE OF THE GIVEN NTUPLE A }

      j := 1;
      WHILE (j <= aSize) and found DO BEGIN
        newMF := FORbit(
          IndexMap@[ CorrectSet@.SetList@.SetElt[ j ] ]
            - V1@.VStart + 1, VertexMap );
        k := 1;
        OtherOrbit := true;
        WHILE (k <= aSize) and OtherOrbit DO BEGIN
          IF (not MatchOrbit[ k ]) and (MFa[ k ] = newMF)
            THEN BEGIN
              OtherOrbit := false;
              MatchOrbit[ k ] := true;
            END ELSE
              k := k + 1;
          END; { WHILE }
        IF OtherOrbit THEN BEGIN

```

```

        { AT LEAST ON OF THE ORBITS OF THE CURRENT SET DID NOT
          MATCH THE ORBITS OF NTUPLE A }

        found := false;
        CorrectSet := CorrectSet@.NextSet
      END ELSE
        j := j + 1
      END { WHILE }
    END ELSE
      CorrectSet := CorrectSet@.NextSet
    END; { WHILE }

    { CREATE A NEW SET ELEMENT FOR THE NTUPLE A }
    { LINK IN TO A SET }

    IF not found THEN BEGIN
      { CREATE A NEW SET FOR NTUPLE A }

      new( CorrectSet );
      CorrectSet@.single := asingle;
      CorrectSet@.SetKind := aSetType;
      CorrectSet@.SetSize := 1;
      CorrectSet@.ntupleSize := aSize;
      CorrectSet@.SetList := nil;
      CorrectSet@.NextSet := Sets[ aSetType ];
      Sets[ aSetType ] := CorrectSet;
      CASE aSetType OF
        Pink: CorrectSet@.TypeTotal := MFa[ 1 ]@.OrbitSize;
        Brown1, GreenBrown1, Green1:
          CorrectSet@.TypeTotal := NChoose2( MFa[1]@.OrbitSize );
        Brown2, GreenBrown2, Green2:
          CorrectSet@.TypeTotal := MFa[1]@.OrbitSize*MFa[2]@.OrbitSize;
        Red1: CorrectSet@.TypeTotal := NChoose3( MFa[1]@.OrbitSize );
        Red3: CorrectSet@.TypeTotal := MFa[1]@.OrbitSize
              * MFa[2]@.OrbitSize * MFa[3]@.OrbitSize;
      Red2: BEGIN
        IF MFa[1] = MFa[2] THEN BEGIN
          i := 1;
          j := 3;
        END ELSE IF MFa[1] = MFa[3] THEN BEGIN
          i := 1;
          j := 2;
        END ELSE BEGIN
          i := 2;
          j := 1;
        END;
        CorrectSet@.TypeTotal := NChoose2( MFa[i]@.OrbitSize )
          * MFa[j]@.OrbitSize
      END
    END { CASE }
  END ELSE
    CorrectSet@.SetSize := CorrectSet@.SetSize + 1;

```

```

a@.NextSetElt := CorrectSet@.SetList;
CorrectSet@.SetList := a;
END; { AddToSets }

{*****}
*   PROCEDURE AddToV2
*
*   This procedure adds vertex q to partition V2 (which is the
*   set of vertices at the next higher distance from the fixed edge
*   than the vertices of partition V1).
*   This involves moving the position of the vertex q in the
*   graph Z (i.e. changing its index).
* {*****}

PROCEDURE AddToV2( q: integer; V2: VTypePtr );

VAR
  qIndex: integer;
  tPtr: GraphEltPtr;

REF
  Z: Graph;
  ZSize: integer;
  IndexMap: IndexMapPtr;

BEGIN
  V2@.VEnd := V2@.VEnd + 1;
  qIndex := IndexMap@[ q ];
  tPtr := Z@[ V2@.VEnd ];
  Z@[ V2@.VEnd ] := Z@[ qIndex ];
  Z@[ qIndex ] := tPtr;
  IndexMap@[ q ] := V2@.VEnd;
  IndexMap@[ tPtr@.Vertex ] := qIndex;
END; { AddToV2 }

{*****}
*   PROCEDURE FindAdj
*
*   Find all vertices in partition V1 which are adjacent to
*   vertex p (which is at index pIndex in graph Z). Return them in
*   ntuple a and the number of them in aSize.
* {*****}

PROCEDURE FindAdj( pIndex: integer; VAR a: ntuple; VAR aSize: integer;
  V1: VTypePtr );

VAR
  i: integer;

REF

```

```

Z: Graph;
ZSize: integer;
IndexMap: IndexMapPtr;

BEGIN
  aSize := 0;
  FOR i := 1 to 3 DO BEGIN
    a[ aSize+1 ] := Z[ pIndex ]@AdjVertex[ i ];
    IF (IndexMap[ a[ aSize+1 ] ] >= V1@VStart) and
       (IndexMap[ a[ aSize+1 ] ] <= V1@VEnd) THEN
      aSize := aSize + 1;
    END; { FOR }
  END; { FindAdj }
END;

{*****}
*
* PROCEDURE AddToKer
*
* A pair of vertices p and q have been found in partition V2
* that are connected to exactly the same vertices in partition
* V1. The transposition (p q) is a generator of the kernel of the
* mapping from the next group to the current group. (The current
* group is AutE of the subgraph Xi of Z. Xi is created by taking
* all vertices of partitions V0 + V1 + V2 + ... + Vi. Xi includes
* the edges of Z induced by these vertices except for those
* between two vertices of Vi. The next group is AutE of the
* subgraph Xi+1 of Z.)
*
* V0 = Vi          V2 = Vi+1
*
{*****}

PROCEDURE AddToKer( p,q: integer; VAR Ker: PtrGenElt );

VAR
  newgen: PtrGenElt;
  i: integer;

REF
  Z: Graph;
  ZSize: integer;
  IndexMap: IndexMapPtr;

BEGIN
  new( newgen );
  newgen@.Gen := ampxnew( ZSize*4 );
  newgen@.GenInverse := ampxnew( ZSize*4 );
  FOR i := 1 to ZSize DO BEGIN
    newgen@.Gen[ i ] := i;

    newgen@.GenInverse[ i ] := i
  END; { FOR }
  newgen@.GenInverse[ p ] := q;
  newgen@.GenInverse[ q ] := p;
  newgen@.Gen[ p ] := q;
  newgen@.Gen[ q ] := p;
  newgen@.NextGenElt := Ker;
  Ker := newgen;
END; { AddToKer }

{*****}
*
* PROCEDURE FindNextPartition
*
* This procedure is passed the current partition V1 consisting
* of all vertices at distance i from the fixed edge. It finds all
* vertices at distance i+1 from the fixed edge (i.e. partition V2)
* by examining vertices adjacent to vertices in V1.
* At the same time, it finds generators for Ker (see the
* documentation for AddToKer) and also the sets of vertices,
* duples of vertices or triples of vertices to be stabilized to
* find G/Ker. Sets of the same type are linked together and the
* top of the list is pointed at by an element of Sets (indexed by
* ntuple type).
*
{*****}

PROCEDURE FindNextPartition( V1: VTypePtr;
  VAR V2: VTypePtr;
  VertexMap: OrbitTypePtr;
  VAR Ker: PtrGenElt;
  VAR Sets: InitSet );

VAR
  a: ntuple; { an ntuple which is part of a set
               of ntuples which should be
               stabilized }
  aIndex: ntuple; { indices in Z of vertices in a }
  aSetType: ntupleType; { the type of the ntuple }
  aSingle: boolean; { is there one (= single) vertex
                     in V2 adjacent to the vertices
                     in ntuple a (and no others in V1)
                     or are there two such vertices? }
  MPa: MPtruple; { ptrs to the orbit representatives
                  (tops of merge-find trees) of the
                  vertices in a }
  aSize: integer; { the number of vertices in a }
  p,q: integer; { vertices in V2 adjacent to a }
  pIndex, qIndex: integer;
  i,j,k: integer;
  NewElt: SetEltPtr;

REF
  Z: Graph;

```

```

ZSize: integer;
IndexMap: IndexMapPtr;

BEGIN
  Ker := nil;

  { INITIALIZE EACH ELEMENT IN SETS TO NIL (NO SETS TO BE
    STABILIZED OF ANY TYPE YET) }

  Sets[ Pink ] := nil;
  Sets[ Brown1 ] := nil;
  Sets[ Brown2 ] := nil;
  Sets[ GreenBrown1 ] := nil;
  Sets[ GreenBrown2 ] := nil;
  Sets[ Green1 ] := nil;
  Sets[ Green2 ] := nil;
  Sets[ Red1 ] := nil;
  Sets[ Red2 ] := nil;
  Sets[ Red3 ] := nil;

  { CREATE A NEW RECORD TO REPRESENT PARTITION V2 }

  new( V2 );
  V2@VStart := V1@VEnd + 1;
  V2@VEnd := V1@VEnd;

  { EXAMINE EACH VERTEX IN V1 IN TURN }

  FOR i := V1@VStart to V1@VEnd DO BEGIN
    aIndex[ 1 ] := i;
    a[ 1 ] := Z[ i ]@Vertex;

    { EXAMINE EACH VERTEX ADJACENT TO THE VERTEX AT INDEX I IN Z }

    FOR j := 1 to 3 DO BEGIN
      p := Z[ i ]@AdjVertex[ j ];
      pIndex := IndexMap@[ p ];
      IF (pIndex > i) and (pIndex <= V1@VEnd) THEN BEGIN

        { THE EDGE GOING FROM A[1] TO P IS AN EDGE WITHIN
          PARTITION V1 AND HASN'T BEEN EXAMINED BEFORE
          (THIS IS A BROWN EDGE -- AN 'ORIGINAL' EDGE) }

        a[ 2 ] := p;
        aIndex[ 2 ] := pIndex;
        Mfa[ 1 ] := Forbit( aIndex[1]-V1@VStart+1, VertexMap );
        Mfa[ 2 ] := Forbit( aIndex[2]-V1@VStart+1, VertexMap );
        a[ 3 ] := 0;
        aIndex[ 3 ] := 0;
        Mfa[ 3 ] := nil;
        aSingle := true;
        aSize := 2;

        { SEE IF IT'S ALSO A GREEN EDGE (I.E. GREENBROWN) -

```

```

          A[1] AND A[2] JOINED TO Q IN V2 AND Q IS NOT ADJACENT
          TO ANY OTHER VERTICES IN V1 }

        q := FindMatch( a, aIndex, 2, V1, V1@VEnd, 0 );
        IF q = 0 THEN BEGIN

          { BROWN EDGE NOT GREENBROWN }

          IF Mfa[ 1 ] = Mfa[ 2 ] THEN
            aSetType := Brown1
          ELSE
            aSetType := Brown2;
          new( NewElt );
          NewElt@SetElt := a;
          AddToSets( NewElt, Mfa, aSetType, aSingle, aSize, Sets,
            VertexMap, V1, IndexMap );
        END ELSE { GREENBROWN EDGE (MAY ALREADY HAVE BEEN SEEN) }
        BEGIN
          IF IndexMap@[ q ] > V2@VEnd THEN BEGIN
            IF Mfa[ 1 ] = Mfa[ 2 ] THEN
              aSetType := GreenBrown1
            ELSE
              aSetType := GreenBrown2;
            AddToV2( q, V2 );
            new( NewElt );
            NewElt@SetElt := a;
            AddToSets( NewElt, Mfa, aSetType, aSingle, aSize, Sets,
              VertexMap, V1, IndexMap );
          END { IF }
        END { ELSE }
      END ELSE IF pIndex > V2@VEnd THEN BEGIN

        { P IS A VERTEX WHICH SHOULD BE ADDED TO V2
          IT MAY BE ADJACENT ONLY TO A[1] IN V1 => PINK
          OR TO TWO VERTICES IN V1 => GREEN EDGE
          OR TO THREE VERTICES IN V1 => RED TRIPLE.
          THERE MAY BE ANOTHER VERTEX Q ADJACENT TO EXACTLY
          THE SAME VERTICES IN V1 AS P. }

        AddToV2( p, V2 );
        pIndex := IndexMap@[ p ];
        FindAdj( pIndex, a, aSize, V1 );
        FOR k := 1 to aSize DO BEGIN
          aIndex[ k ] := IndexMap@[ a[ k ] ];
          Mfa[ k ] := Forbit( aIndex[k]-V1@VStart+1,
            VertexMap );
        END; { FOR }
        q := FindMatch( a, aIndex, aSize, V1, V2@VEnd, p );
        IF q <> 0 THEN BEGIN
          aSingle := false;
          AddToV2( q, V2 );
          AddToKer( p, q, Ker );
        END ELSE
          aSingle := true;

```

```

CASE aSize OF
1: aSetType := Pink;
2: BEGIN
    IF Mfa[ 1 ] = Mfa[ 2 ] THEN
        aSetType := Green1
    ELSE
        aSetType := Green2;
    IF aSingle THEN
        FOR k := 1 to 3 DO BEGIN
            IF Z@[ aIndex[ 1 ] ]@.AdjVertex[ k ]
                = a[ 2 ] THEN
                IF Mfa[ 1 ] = Mfa[ 2 ] THEN
                    aSetType := GreenBrown1
                ELSE
                    aSetType := GreenBrown2
                END; { FOR }
        END; { CASE 2 }
    OTHERWISE IF (Mfa[1] = Mfa[2]) and (Mfa[1] = Mfa[3]) THEN
        aSetType := Red1
    ELSE IF (Mfa[1] <> Mfa[2]) and (Mfa[1] <> Mfa[3])
        and (Mfa[2] <> Mfa[3]) THEN
        aSetType := Red3
    ELSE
        aSetType := Red2
    END; { CASE STMT }
    new( NewElt );
    NewElt@.SetElt := a;
    AddToSets( NewElt, Mfa, aSetType, aSingle, aSize, Sets,
        VertexMap, V1, IndexMap )
    { ELSE }
END { FOR }
END; { FOR }
END; { FindNextPartition }

```

```

{*****}
* Segment MForbit *
* *
* This segment contains the routines which perform a merge/find *
* algorithm to find the orbits of the vertices in partition V1. *
* *****}
{*****}
* FUNCTION Forbit *
* *
* Given the index of a the pointer to a vertex's merge-find *
* element in VertexMap, find and return a pointer to the top of *
* the merge-find tree of the orbit that the vertex is in. *
* *
* Each merge-find element on the path from the given vertex's *
* merge-find element to the top of the tree is altered to point *
* at the top of the tree so that future Forbits require less *
* *****}

```

```

* time. *
* *****}
FUNCTION Forbit( i: integer; { index in VertexMap of the pointer
                             to a vertex's merge-find element }
    VertexMap: OrbitTypePtr ): MForbitPtr;

VAR
    s,t,u: MForbitPtr;

REF
    IndexMap: IndexMapPtr;
    Z: Graph;
    ZSize: integer;

BEGIN
    { FIND THE TOP OF THE MERGE-FIND TREE }

    s := VertexMap@[i];
    t := s;
    WHILE not s@.TopVertex DO
        s := s@.ParentVertex;
    END;

    { PATH COMPRESSION TO REDUCE FIND TIME }

    IF t <> s THEN
        WHILE t@.ParentVertex <> s DO BEGIN
            u := t@.ParentVertex;
            t@.ParentVertex := s;
            t := u;
        END; { WHILE }
    END;
    Forbit := s;
END; { Forbit }

{*****}
* PROCEDURE MergeOrbit *
* *
* Given pointers to the tops of 2 merge-find trees representing *
* 2 orbits, merge them by making the top of the larger tree the *
* parent of the smaller tree and merging the circular doubly- *
* linked lists of vertices in the orbits. *
* *****}

PROCEDURE MergeOrbit( MF1, MF2: MForbitPtr );

VAR
    temp: MForbitPtr;
    OE1,OE2: OrbitEltPtr;

REF

```

```

IndexMap: IndexMapPtr;
Z: Graph;
ZSize: integer;

BEGIN
  { MAKE SURE MF1 POINTS AT THE ROOT OF THE LARGER TREE }
  IF MF1@.OrbitSize < MF2@.OrbitSize THEN BEGIN
    Temp := MF1;
    MF1 := MF2;
    MF2 := Temp;
  END; { IF }

  { MERGE MF2 (THE SMALLER ORBIT) INTO MF1 }
  OE1 := MF1@.OrbitList@.OBLink;
  OE2 := MF2@.OrbitList@.OBLink;
  MF1@.OrbitList@.OBLink := OE2;
  OE1@.OFLink := MF2@.OrbitList;
  MF2@.OrbitList@.OBLink := OE1;
  MF2@.OFLink := MF1@.OrbitList;
  MF1@.OrbitSize := MF1@.OrbitSize + MF2@.OrbitSize;
  MF2@.TopVertex := false;
  MF2@.ParentVertex := MF1;
  MF2@.MergeOrbit;
END;

{ ***** }
*
*   PROCEDURE Forbits
*
*   Find the orbits of the vertices in partition V1 (vertices at
*   distance i from the fixed edge). This is done using a merge-
*   find algorithm:
*
*   FOR EACH vertex v in partition V1 DO
*   FOR EACH generator q of the group G DO
*     x := ( v )q;
*     IF x and v are not in the same merge-find tree THEN
*       merge the two trees
*
*   { ***** }
PROCEDURE Forbits( V1: VTypePtr; VertexMap: OrbitTypePtr;
  G: PtrGenElt );

VAR
  i: integer;
  MF1, MF2: MForbitPtr;
  q: PtrGenElt;
  VMSize: integer;
  OE: OrbitEltPtr;
  x: pt;

```

```

REF
IndexMap: IndexMapPtr;
Z: Graph;
ZSize: integer;

BEGIN
  FOR i := V1@.VStart to V1@.VEnd DO BEGIN
    q := G;
    WHILE q <> nil DO BEGIN
      x := q@.gen@[ Z@[ i ]@.vertex ];
      MF1 := Forbit( i-V1@.VStart+1, VertexMap );
      MF2 := Forbit( IndexMap@[x]-V1@.VStart+1, VertexMap );
      IF MF1 <> MF2 THEN
        MergeOrbit( MF1, MF2 );
      q := q@.NextGenElt;
    END { WHILE };
  END { FOR };
END; { Forbits }

{ ***** }
*
*   PROCEDURE CreateVertexMap
*
*   { ***** }
PROCEDURE CreateVertexMap( V1: VTypePtr; VAR VertexMap: OrbitTypePtr );

VAR
  VMSize: integer;
  i: integer;
  MF1: MForbitPtr;
  OE: OrbitEltPtr;

REF
IndexMap: IndexMapPtr;
Z: Graph;
ZSize: integer;

BEGIN
  { CREATE THE MAPPING FROM VERTEX TO MERGE-FIND ELEMENT }
  VMSize := V1@.VEnd - V1@.VStart + 1;
  VertexMap := ampxnew( VMSize*4 );
  FOR i := 1 to VMSize DO BEGIN
    new( MF1 );
    VertexMap@[ i ] := MF1;
    MF1@.TopVertex := true;
    MF1@.OrbitSize := 1;
    new( OE );
    OE@.OVertex := Z@[ i + V1@.VStart - 1 ]@.Vertex;
    OE@.OFLink := OE;
    OE@.OBLink := OE;
    MF1@.OrbitList := OE
  
```

```

END; { FOR }
END; { CreateVertexMap }
*****
PROCEDURE ReOrderMForbits
*
*   Reset the merge-find structures to make each vertex into its
*   own orbit so that the new orbits can be found.
*
*****
PROCEDURE ReOrderMForbits( VertexMap: OrbitTypePtr; V1: VTypePtr );

VAR
  i, j: integer;
  s, t: OrbitEltPtr;

REF
  IndexMap: IndexMapPtr;
  Z: Graph;
  ZSize: integer;

BEGIN
  FOR i := 1 to V1@.VEnd-V1@.VStart+1 DO
    IF VertexMap@[i]@.TopVertex THEN BEGIN
      IF VertexMap@[i]@.OrbitSize <> 1 THEN BEGIN
        s := VertexMap@[i]@.OrbitList;
        s@.OBLink@.OFLink := nil;
        WHILE s <> nil DO BEGIN
          t := s@.OFLink;
          j := IndexMap@[ s@.OVertex ] - V1@.VStart + 1;
          VertexMap@[j]@.TopVertex := true;
          VertexMap@[j]@.OrbitSize := 1;
          VertexMap@[j]@.OrbitList := s;
          s@.OFLink := s;
          s@.OBLink := s;
          s := t;
        END { WHILE }
      END { IF }
    END { IF }
  END; { ReOrderOrbits }
*****
*
*   Segment BlkSys (Block System)
*
*   This segment contains the routines which find a block system
*   containing two blocks for the 2-group G and the subgroup P in G
*   which fixes the block system. P is normal in G and has index 2
*   in G.
*
*****

```

```

SEGMENT BlkSys;

CONST
  hugesize = 10000;

TYPE
  { Merge/find pointers used to find the orbits of pairs of ntuples.
    This is used to find an orbit which, when treated as a graph,
    has exactly two components. The two sets of vertices of the
    components (sets of ntuples) are a block system for the group. }

  SetPairsTypePtr = @SetPairsType;
  MFPairPtr = @MFPair;
  PairPtr = @PairElt;
  SetPairsType = array[ 1..hugesize ] of MFPairPtr;
  { MAPS PAIRS OF NTUPLES TO THEIR
    MERGE/FIND POINTER }

  MFPair = RECORD
    CASE TopPair: boolean OF
      false: (Parent: MFPairPtr);
      true: (Size: integer;
             PairList: PairPtr;
             FSet, BSet: MFPairPtr)
    END;
  PairElt = RECORD
    N1, N2: integer;
    FPair, BPair: PairPtr
  END;

  { MERGE/FIND POINTERS USED TO FIND THE COMPONENTS OF AN
    ORBIT OF PAIRS OF NTUPLES. }

  SetComponentTypePtr = @SCType;
  MFBblockPtr = @MFBblock;
  SCType = array[ 1..hugesize ] of MFBblockPtr;
  MFBblock = RECORD
    CASE TopBlock: boolean OF
      false: (parent: MFBblockPtr );
      true: ( size: integer;
              blocknum: integer )
    END;
  END;

  { SOME ROUTINES WERE NOT INCLUDED IN THIS LISTING:

  DispSP: dispose of the merge/find structure used to find orbits
    of pairs of ntuples.
  DispSC: dispose of the merge/find structure used to find the
    components of the graph associated with an orbit of pairs
    of ntuples.
  DispP: dispose of the merge/find structure used to find blocks of
    blocks.
  SortNtuple: sorts an ntuple.
  }

```



```

*****
* FUNCTION FindComponent
*
* This function follows the merge/find pointers from the node
* representing the ntuple at index i in an ordered array of all
* of the current type of ntuples to the root of the merge/find
* tree. The tree represents a component of the graph induced by
* an orbit of pairs of ntuples treated as edges with the ntuples
* being the vertices. A pointer to the root of the tree is
* returned.
*
*****
FUNCTION FindComponent( i: integer; SetComponents: SetComponentTypePtr)
;MFBBlockPtr;

VAR
  MFa, MFb, MFc: MFBBlockPtr;

BEGIN
  MFa := SetComponents@[ i ];
  WHILE not MFa.TopBlock DO MFa := MFa.Parent;
  IF MFa <> SetComponents@[ i ] THEN BEGIN
    MFb := SetComponents@[ i ];
    WHILE MFb.Parent <> MFa DO BEGIN
      MFc := MFb.Parent;
      MFb.Parent := MFa;
      MFb := MFc;
    END;
  END; { IF }
  FindComponent := MFa;
END; { FindComponent }

*****
* FUNCTION SearchON
*
* This procedure searches an ordered array of ntuples (ON) for
* a particular ntuple (p). If the ntuple is not found, then 0 is
* returned. Otherwise, the index of p in ON is returned.
*
*****
FUNCTION SearchON( VAR p: ntuple; ON: SetTypePtr; ONSize: integer;
ntupleSize: integer ): integer;

VAR
  top, bot, mid: integer;
  done: boolean;
  i: integer;
  found: boolean;

```

```

BEGIN
  Sortntuple( p, ntupleSize );
  top := ONSize;
  bot := 1;
  found := false;
  WHILE top >= bot DO BEGIN
    mid := (top + bot) div 2;
    i := 1;
    done := false;
    WHILE (i <= ntupleSize) and (not done) DO
      IF p[ i ] <> ON[ mid, i ] THEN
        done := true
      ELSE
        i := i + 1;
    IF i > ntupleSize THEN BEGIN
      bot := top + 1;
      found := true;
    END ELSE
      IF p[ i ] > ON[ mid, i ] THEN
        bot := mid + 1
      ELSE
        top := mid - 1;
    END; { WHILE }
    IF found THEN
      SearchON := mid;
    ELSE
      SearchON := 0;
    END; { SearchON }
  END;

*****
* FUNCTION StabBlock
*
* This function returns the stabilizer of the block containing
* the first ntuple (in order). The stabilizer is returned as a
* tower of point stabilizers created by Schreier's algorithm.
* Since there are only two blocks, the size of the orbit of the
* first block is at most 2.
*
* VARIABLE DICTIONARY
*
* G: a list of generators of the group we want to find the block
* stabilizer in
* SetComponents: maps an ntuple to its merge/find pointer used to
* find which block (component) it's in
* ON: an orbit of ntuples that G is transitive on
* TypeTotal: the number of ntuples in the orbit ON
* ntupleSize: the number of vertices in the ntuple
* B2TransElt: a pointer to the transversal element for the 2nd
* block
* H: the stabilizer of the first block
* B2: the index in ON of the representative ntuple of block 2
* MFB1, MFB2: pointers to the tops of the merge/find trees for
* blocks one and two, respectively
*
*****

```

```

*****
FUNCTION StabBlock( G: PtrGenElt; SetComponents: SetComponentTypePtr;
ON: SetTypePtr; TypeTotal: integer;
ntupleSize: integer ): Group;

```

```

VAR
  B2TransElt: PtrTransElt;
  H: Group;
  base: baseArray;
  baseIndex: integer;
  Q, R: PtrGenElt;
  MapB: ntuple;
  B2: integer;
  MFB1, MFB2, MFBMapB: MFBblockPtr;
  i, j, k: integer;
  newTransElt: boolean;
  qp, gp: permutation;

```

```

REF
  ZSize: integer;

```

```

BEGIN
  Q := G;
  b2TransElt := nil;
  newTransElt := false;
  base := ampXnew( ZSize*4 );
  FOR i := 1 to ZSize DO base@[ i ] := i;
  baseIndex := 2;
  H := IGroup( 1, ZSize );
  MFB1 := FindComponent( 1, SetComponents );

  { APPLY EACH OF THE GENERATORS OF G IN TURN TO THE
    ORBIT OF BLOCK 1 }

  WHILE Q <> nil DO BEGIN
    FOR i := 1 to ntupleSize DO
      MapB[ i ] := Q@.Gen@[ ON@[ 1, i ] ];
      j := SearchON( MapB, ON, TypeTotal, ntupleSize );
      MFBMapB := FindComponent( j, SetComponents );
      IF (MFBMapB <> MFB1) and (B2TransElt = nil) THEN BEGIN
        { GENERATOR Q MAPPED BLOCK 1 TO BLOCK 2. BLOCK 2
          IS NOT CURRENTLY IN THE ORBIT OF BLOCK 1, SO ADD
          IT (I.E. CREATE IT'S TRANSVERSAL ELEMENT AND
          POINT B2TransElt AT THE TRANSVERSAL ELEMENT). }

        newTransElt := true;
        B2 := j;
        MFB2 := MFBMapB;
        new( B2TransElt );
        B2TransElt@a := ampXnew( ZSize*4 );

        B2TransElt@.ainverse := ampXnew( ZSize*4 );
        FOR j := 1 to ZSize DO BEGIN
          B2TransElt@a@[ j ] := Q@.Gen@[ j ];
          B2TransElt@.ainverse@[ j ] := Q@.GenInverse@[ j ];
        END;
      END ELSE
      IF MFBMapB = MFB1 THEN BEGIN
        { GENERATOR Q MAPPED BLOCK 1 TO BLOCK 1, WHICH
          GIVES US A NEW ELEMENT OF STAB(BLOCK 1) }

        IF not GroupElement( Q@.Gen, H ) THEN
          AddGen( Q@.Gen, H, base, BaseIndex );
        END ELSE BEGIN
          { GENERATOR Q MAPPED BLOCK 1 TO BLOCK 2. BLOCK 2 IS
            ALREADY IN THE ORBIT OF BLOCK 1, SO WE CAN ADD A
            NEW ELEMENT TO STAB(BLOCK 1). }

          gp := MultPerms( Q@.Gen, B2TransElt@.ainverse, ZSize );
          IF not GroupElement( gp, H ) THEN
            AddGen( gp, H, base, BaseIndex );
          dispose( gp );
        END; { ELSE }

        { IF BLOCK 2 IS IN THE ORBIT OF BLOCK 1, APPLY GENERATOR
          Q TO BLOCK 2. }

        IF B2TransElt <> nil THEN BEGIN
          FOR i := 1 to ntupleSize DO
            MapB[ i ] := Q@.Gen@[ ON@[ B2, i ] ];
            j := SearchON( MapB, ON, TypeTotal, ntupleSize );
            MFBMapB := FindComponent( j, SetComponents );
            IF MFBMapB = MFB1 THEN BEGIN
              { Q MAPPED BLOCK 2 TO BLOCK 1, SO A NEW ELEMENT OF
                STAB(BLOCK 1) HAS BEEN FOUND. }

              gp := MultPerms( B2TransElt@a, Q@.Gen, ZSize );
              IF not GroupElement( gp, H ) THEN
                AddGen( gp, H, base, BaseIndex );
              dispose( gp );
            END ELSE BEGIN
              { Q MAPPED BLOCK 2 TO BLOCK 2, WHICH GIVES US A
                NEW ELEMENT OF STAB(BLOCK 1). }

              gp := MultPerms( B2TransElt@a, Q@.Gen, ZSize );
              gp := MultPerms( gp, B2TransElt@.ainverse, ZSize );
              dispose( gp );
              IF not GroupElement( gp, H ) THEN
                AddGen( gp, H, base, BaseIndex );
              dispose( gp );
            END; { ELSE }
          END;
        END;
      END;
    END;
  END;

```

```

{ IF BLOCK 2 HAS JUST BEEN ADDED TO THE ORBIT OF BLOCK 1
  APPLY ALL PREVIOUSLY PROCESSED GENERATORS OF G TO
  BLOCK 2.
}

IF newTransElt THEN BEGIN
  newTransElt := false;
  R := G;
  WHILE R <> Q DO BEGIN
    FOR i := 1 to ntupleSize DO
      MapB[ i ] := R@.Gen@[ ON@[ B2, i ] ];
      j := SearchON( MapB, ON, TypeTotal, ntupleSize );
      MFMapB := FindComponent( j, SetComponents );
      IF MFMapB = MFB1 THEN BEGIN
        { GENERATOR R MAPPED BLOCK 1 TO BLOCK 2 }

        gp := MultPerms( B2TransElt@a, R@.Gen, ZSize );
        IF not GroupElement( gp, H ) THEN
          AddGen( gp, H, base, BaseIndex );
        dispose( gp );
      END ELSE BEGIN
        { GENERATOR R MAPPED BLOCK 2 TO BLOCK 2 }

        qp := MultPerms( B2TransElt@a, R@.Gen, ZSize );
        qp := MultPerms( qp, B2TransElt@.ainverse, ZSize );
        dispose( qp );
        IF not GroupElement( qp, H ) THEN
          AddGen( qp, H, base, BaseIndex );
        dispose( qp );
      END;
      { ELSE }
      R := R@.NextGenElt;
    END; { WHILE }
  END; { IF }
  Q := Q@.NextGenElt;
END; { WHILE }
dispose( base );
dispose( B2TransElt@a ); dispose( B2TransElt@.ainverse );
dispose( B2TransElt );
StabBlock := H
END; { StabBlock }

{*****}
*
* PROCEDURE CreateSetPairs
*
* Create the merge/find data structure used to find orbits of
* pairs of ntuples. The structure is set up so that each pair of
* ntuples is in its own orbit.
*
* {*****}

```

```

PROCEDURE CreateSetPairs( ONSize: integer; Total: integer;
  VAR SetPairs: SetPairsTypePtr;
  VAR FirstSet: MFPairPtr );

```

```

VAR
  i, j, k, l: integer;
  MFp: MFPairPtr;
  pPairElt: PairPtr;

BEGIN
  SetPairs := ampxnew( Total*4 );
  FirstSet := nil;
  k := 0;
  FOR i := 1 to (ONSize - 1) DO BEGIN
    FOR j := i+1 to ONSize DO BEGIN
      new( MFp );
      k := k + 1;
      SetPairs@[ k ] := MFp;
      MFp@.TopPair := true;
      MFp@.Size := 1;
      MFp@.FSet := FirstSet;
      MFp@.BSet := nil;
      IF FirstSet <> nil THEN
        FirstSet@.BSet := MFp;
      FirstSet := MFp;
      new( pPairElt );
      MFp@.PairList := pPairElt;
      pPairElt@.N1 := i;
      pPairElt@.N2 := j;
      pPairElt@.FPair := pPairElt;
      pPairElt@.BPair := pPairElt;
    END; { FOR }
  END; { FOR }
END; { CreateSetPairs }

```

```

{*****}
*
* FUNCTION FSetPrs
*
* This function is passed an index into the mapping of a pair
* of ntuples to the pair's merge/find pointer. The merge/find
* tree of the pair is traversed until the root is found. The path
* traversed is then compressed so that every node on the path
* points at the root of the tree. A pointer to the root of the
* tree (which represents the orbit of pairs of ntuples) is
* returned.
*
* {*****}

```

```

FUNCTION FSetPrs( a: integer; SetPairs: SetPairsTypePtr ):
  MFPairPtr;

```

```

VAR
  MFA, MFB, MFC: MFPairPtr;

```

```

BEGIN
  Mfa := SetPairs@[ a ];
  WHILE not Mfa.TopPair DO
    Mfa := Mfa.Parent;
  IF not SetPairs@[ a ]@.TopPair THEN BEGIN
    Mfb := SetPairs@[ a ];
    WHILE Mfb.Parent <> Mfa DO BEGIN
      Mfc := Mfb.Parent;
      Mfb.Parent := Mfa;
      Mfb := Mfc;
    END; { WHILE }
  END; { IF }
  FSetPrs := Mfa
END; { FSetPrs }

{*****}
*
*   PROCEDURE MergeSetPairs
*
*   This procedure is passed the pointers to the tops of two
*   merge/find trees which are used to find the orbits (sets) of
*   pairs of ntuples. The two orbits are merged by merging the
*   lists of elements the two trees and then pointing the root of
*   the smaller tree at the root of the larger tree.
*
*   {*****}

PROCEDURE MergeSetPairs( Mfa, Mfb: MFPairPtr; FirstSet: MFPairPtr );

VAR
  Mfc: MFPairPtr;
  aPair, bPair: PairPtr;

BEGIN
  IF Mfa.Size < Mfb.Size THEN BEGIN
    Mfc := Mfa;
    Mfa := Mfb;
    Mfb := Mfc;
  END;
  IF FirstSet = Mfb THEN FirstSet := Mfb.FSet;
  Mfa.Size := Mfa.Size + Mfb.Size;
  IF Mfb.FSet <> nil THEN
    Mfb.FSet@.BSet := Mfb.BSet;
  IF Mfb.BSet <> nil THEN
    Mfb.BSet@.FSet := Mfb.FSet;
  aPair := Mfa.PairList@.BPair;
  bPair := Mfb.PairList@.BPair;
  Mfa.PairList@.BPair := bPair;
  bPair.FPair := Mfa.PairList;
  Mfb.PairList@.BPair := aPair;
  aPair.FPair := Mfb.PairList;
  Mfb.TopPair := false;
  Mfb.Parent := Mfa;

END; { MergeSetPairs }

{*****}
*
*   PROCEDURE CreateSetComponents
*
*   This procedure creates the merge/find data structure used to
*   find the components of the graph which is created by an orbit
*   of pairs of ntuples. The structure is set up so that each
*   ntuple is in its own component.
*
*   {*****}

PROCEDURE CreateSetComponents( VAR SetComponents: SetComponentTypePtr;
                               TypeTotal: integer );

VAR
  i: integer;
  MF1: MFBlockPtr;

BEGIN
  SetComponents := ampxnew( TypeTotal*4 );
  FOR i := 1 to TypeTotal DO BEGIN
    new( MF1 );
    MF1.TopBlock := true;
    MF1.Size := 1;
    MF1.BlockNum := 0;
    SetComponents@[ i ] := MF1;
  END; { FOR }
END; { CreateSetComponents }

{*****}
*
*   PROCEDURE MergeComponents
*
*   Merge two merge/find trees which represent a set of ntuples
*   which are the vertices in components of a graph.
*
*   {*****}

PROCEDURE MergeComponents( MF1, MF2: MFBlockPtr );

VAR
  MF3: MFBlockPtr;

BEGIN
  IF MF1.Size < MF2.Size THEN BEGIN
    MF3 := MF1;
    MF1 := MF2;
    MF2 := MF3;
  END;
  MF1.Size := MF1.Size + MF2.Size;
  MF2.TopBlock := false;
  MF2.Parent := MF1;

```

```

END; { MergeComponents }

{*****}
*
*   PROCEDURE ReOrderSetComponents
*
*       Reset the merge/find pointers back to the initial state where
*       every ntuple is in its own component so that the merge/find
*       pointers can be reused.
*
*   {*****}
PROCEDURE ReOrderSetComponents( SetComponents: SetComponentTypePtr;
                               TypeTotal: integer );

VAR
    i: integer;

BEGIN
    FOR i := 1 to TypeTotal DO BEGIN
        SetComponents[i].TopBlock := true;
        SetComponents[i].Size := 1;
        SetComponents[i].BlockNum := 0;
    END; { FOR }
END; { ReOrderSetComponents }

{*****}
*
*   PROCEDURE Iterate
*
*       A set of 2**k blocks have been found where k>1. We want 2
*       blocks. What we do is to find blocks of blocks. This is done
*       by applying a different orbit of pairs of ntuples than the one
*       used to find this block system. Keep applying other orbits of
*       pairs of ntuples that don't cause ALL the blocks to merge but do
*       cause the number of blocks to go down until we have found a
*       system of 2 blocks.
*
*   {*****}
PROCEDURE Iterate( SetComponents: SetComponentTypePtr;
                  VAR numComponents: integer; ON: SetTypePtr; TypeTotal: integer;
                  SetPairSH: MFPPtr; ntupleSize: integer);

TYPE
    { MERGE/FIND DATA STRUCTURE USED TO FIND BLOCKS OF BLOCKS }

    PTypePtr = @PType;
    MFPPtr = @MFP;
    PType = array[ 1..hugeSize ] of MFPPtr;
    MFP = RECORD
        CASE TopP: boolean OF
            false: ( parent: MFPPtr );
            true: ( size: integer;
                   BlockNum: integer );
        END;

VAR
    Q: MFPPtr;
    HP: PairPtr;
    X: PTypePtr;
    numBlocks: integer;
    i1, i2: integer;
    MFB1, MFB2: MFBblockPtr;
    MFP1, MFP2: MFPPtr;
    Count: integer;

{*****}
*
*   PROCEDURE MergeP
*
*       This procedure is passed two pointers to the roots of two
*       merge/find trees used to find blocks of blocks. The smaller
*       tree is merged into the larger tree.
*
*   {*****}
PROCEDURE MergeP( MF1, MF2: MFPPtr );

VAR
    MF3: MFPPtr;

BEGIN
    IF MF1.Size < MF2.Size THEN BEGIN
        MF3 := MF1;
        MF1 := MF2;
        MF2 := MF3;
    END;
    MF1.Size := MF1.Size + MF2.Size;
    MF2.TopP := false;
    MF2.Parent := MF1;
END; { MergeP }

{*****}
*
*   PROCEDURE CreateX
*
*       This procedure initializes the data structure X which maps
*       blocks to their merge/find pointers. Each block is initially
*       in its own merge/find tree.
*
*   {*****}
PROCEDURE CreateX( numComponents: integer; VAR X: PTypePtr );

VAR

```

```

i: integer;
MFP1: MFPPtr;

BEGIN
  X := ampxnew( numComponents*4 );
  FOR i := 1 to numComponents DO BEGIN
    new( MFP1 );
    MFP1@.TopP := true;
    MFP1@.Size := 1;
    X@i := MFP1;
  END; { FOR }
END; { CreateX }

{*****}
*
* FUNCTION FindP
*
* This function follows parent pointers to find the root of
* the merge/find tree containing block i. Once the root is found
* the path from the merge/find pointer for block i to the root is
* compressed so that all merge/find pointers along the path point
* directly at the root.
*
{*****}

FUNCTION FindP( i: integer; X: PTypePtr ): MFPPtr;

VAR
  MFP1, MFP2, MFP3: MFPPtr;

BEGIN
  MFP1 := X@i;
  WHILE not MFP1@.TopP DO
    MFP1 := MFP1@.Parent;
  IF MFP1 <> X@i THEN BEGIN
    MFP2 := X@i;
    WHILE MFP2@.Parent <> MFP1 DO BEGIN
      MFP3 := MFP2@.Parent;
      MFP2@.Parent := MFP1;
      MFP2 := MFP3;
    END; { WHILE }
    END; { IF }
    FindP := MFP1;
  END; { FindP }

{*****}
*
* MAIN LINE OF ITERATE
*
{*****}

BEGIN { Iterate }
  Q := SetPairSH;

```

```

Count := 0;

{ APPLY EACH ORBIT Q OF PAIRS OF NTUPLES UNTIL WE HAVE 2 BLOCKS }

WHILE (Q <> nil) and (numComponents <> 2) DO BEGIN
  HP := Q@.PairList;

  { FIRST APPLY THE ORBIT OF PAIRS OF NTUPLES TO THE BLOCKS OF
    BLOCKS MERGE/FIND DATA STRUCTURE TO SEE IF IT REDUCES THE
    NUMBER OF BLOCKS WITHOUT MERGING THEM ALL INTO ONE BLOCK }

  CreateX( numComponents, X );
  numBlocks := numComponents;
  REPEAT
    MFB1 := FindComponent( HP@.N1, SetComponents );
    IF MFB1@.BlockNum = 0 THEN BEGIN
      Count := Count + 1;
      MFB1@.BlockNum := Count;
    END;
    MFB2 := FindComponent( HP@.N2, SetComponents );
    IF MFB2@.BlockNum = 0 THEN BEGIN
      Count := Count + 1;
      MFB2@.BlockNum := Count;
    END; { IF }
    MFP1 := FindP( MFB1@.BlockNum, X );
    MFP2 := FindP( MFB2@.BlockNum, X );
    IF MFP1 <> MFP2 THEN BEGIN
      MergeP( MFP1, MFP2 );
      numBlocks := numBlocks - 1;
    END; { IF }
    HP := HP@.FPair;
  UNTIL (HP = Q@.PairList);

  { IF THE NUMBER OF BLOCKS WAS REDUCED (BUT NOT INTO JUST ONE
    BLOCK) THEN APPLY THE ORBIT OF PAIRS OF NTUPLES TO THE
    MERGE/FIND DATA STRUCTURE WHICH FINDS BLOCKS OF NTUPLES }

  IF (numBlocks > 1) and (numBlocks < numComponents) THEN BEGIN
    HP := Q@.PairList;
    numBlocks := numComponents;
    REPEAT
      MFB1 := FindComponent( HP@.N1, SetComponents );
      MFB2 := FindComponent( HP@.N2, SetComponents );
      IF MFB1 <> MFB2 THEN BEGIN
        MergeComponents(MFB1, MFB2 );
        numComponents := numComponents - 1;
      END;
      HP := HP@.FPair;
    UNTIL (HP = Q@.PairList );
    FOR i1 := 1 to TypeTotal DO
      IF SetComponents@i1@.TopBlock THEN
        SetComponents@i1@.BlockNum := 0;
    Count := 0;
  END; { IF }

```

```

IF numComponents <> 2 THEN
  Q := Q@FSet;
  DispP( X, numBlocks );
END; { WHILE }
END; { Iterate }

*****
FUNCTION BlkSys
*
* G is transitive on the orbit of ntuples of the current type.
* This function returns an intransitive subgroup H of index 2 in G
* (G is a 2-group). This is done by finding a block system of
* ntuples containing two blocks and then setting H to the stab-
* ilizer of one of the blocks. The block system is found by
* looking at G2 acting on pairs of ntuples. If the ntuples are
* treated as vertices of graphs and the edges are pairs of ntuples
* within an orbit, the components are the blocks. A graph with
* two components is found using merge/find pointers.
*
*****
FUNCTION BlkSys( G: PtrGenElt; ntupleSize: integer;
  ON: SetTypePtr; ONSize: integer): Group;

VAR
  SetPairs: SetPairTypePtr; { mapping from pairs of ntuples
    to their merge/find pointers
    used to find orbits of pairs
    of ntuples }
  FirstSet: MFPairPtr; { pointer to a linked list of sets
    of pairs of ntuples (i.e. to
    the roots of the merge/find
    trees representing the sets }

  r: PtrGenElt;
  pMap, qMap: ntuple;
  i, j, k: integer;
  qMapPosn, pMapPosn: integer;
  x, y, x1: integer;
  xMF, yMF: MFPairPtr;
  SetComponents: SetComponentTypePtr;
  { mapping from ntuple to its
    merge/find pointer used to
    find blocks }

  numComponents: integer;
  SetPairSH: MFPairPtr; { ptr to the root of a merge/find
    tree of pairs of ntuples.
    used to apply each orbit of
    pairs of ntuples to the
    merge/find structure SetComponents
    which is used to find a block
    system. }

  HPair: PairPtr;
  MF1, MF2: MFBLOCKPtr;

  H: Group;
  GSet: MFPairPtr; { a linked list of orbits of
    pairs of ntuples which were
    disconnected but had more than
    two components. }

  PQ: MFPairPtr;

REF
  ZSize: integer;

BEGIN
  IF ONSize = 2 THEN
    H := IGroup( 1, ZSize )
  ELSE BEGIN
    { FIND THE ORBITS (SETS) OF PAIRS OF NTUPLES SO THAT WE CAN
      FIND A BLOCK SYSTEM }

    x1 := NChoose2( ONSize );
    CreateSetPairs( ONSize, x1, SetPairs, FirstSet );
    x := 0;
    FOR i := 1 to (ONSize - 1) DO BEGIN
      FOR j := (i + 1) to ONSize DO BEGIN
        x := x + 1;
        r := G;
        WHILE r <> nil DO BEGIN
          { MAP THE PAIR OF NTUPLES AT POSITIONS I AND J
            IN ORBIT OF NTUPLES ON }

          FOR k := 1 to ntupleSize DO BEGIN
            pMap[ k ] := r@Gen@[ ON@[ i, k ] ];
            qMap[ k ] := r@Gen@[ ON@[ j, k ] ];
          END;
          pMapPosn := SearchON( pMap, ON, ONSize, ntupleSize );
          qMapPosn := SearchON( qMap, ON, ONSize, ntupleSize );
          IF qMapPosn < pMapPosn THEN
            y := x1 - (NChoose2(ONSize - qMapPosn) + ONSize -
              pMapPosn)
          ELSE
            y := x1 - (NChoose2(ONSize - pMapPosn) + ONSize -
              qMapPosn);
          xMF := FSetPrs( x, SetPairs );
          yMF := FSetPrs( y, SetPairs );
          IF xMF <> yMF THEN
            MergeSetPairs( xmf, yMF, FirstSet );
          r := r@NextGenElt;
        END; { WHILE }
      END; { FOR }
    END; { FOR }

    { TO FIND A BLOCK SYSTEM WITH 2 BLOCKS:
      - TREAT THE NTUPLES AS VERTICES AND THE PAIRS OF NTUPLES
        WITHIN AN ORBIT OF PAIRS AS EDGES
    }
  END;

```



```

- SEE IF THE RESULTING GRAPH HAS 2 COMPONENTS
}

CreateSetComponents( SetComponents, ONSize );
GSet := nil;
numComponents := ONSize;
SetPairSH := FirstSet;
WHILE (numComponents <> 2) and (SetPairSH <> nil) DO BEGIN
  HPair := SetPairSH.PairList;
  REPEAT
    MF1 := FindComponent( HPair@.N1, SetComponents );
    MF2 := FindComponent( HPair@.N2, SetComponents );
    IF MF1 <> MF2 THEN BEGIN
      MergeComponents( MF1, MF2 );
      numComponents := numComponents - 1;
    END;
  UNTIL (HPair = SetPairSH.PairList);
  IF numComponents <> 2 THEN BEGIN
    PQ := SetPairSH@.FSet;
    IF numComponents > 2 THEN BEGIN
      { add this orbit of pairs of ntuples to a list
        that may help to find a block system by
        iterating on blocks of blocks if we
        do not find a block system with 2 blocks
        this way }

      IF PQ <> nil THEN
        PQ@.BSet := SetPairSH@.BSet;
      IF SetPairSH@.BSet <> nil THEN BEGIN
        SetPairSH@.BSet@.FSet := PQ;
        SetPairSH@.BSet := nil;
      END;
      IF PQ <> nil THEN
        SetPairSH@.FSet := GSet;
      IF GSet <> nil THEN
        GSet@.BSet := SetPairSH;
        GSet := SetPairSH;
      END;
      IF PQ <> nil THEN
        SetPairSH := PQ;
        ReOrderSetComponents( SetComponents, ONSize );
        numComponents := ONSize;
      END;
    END;
  END;
  { IF WE DIDN'T GET TWO COMPONENTS, WE NEED TO FIND BLOCKS
    OF BLOCKS, I.E. ITERATE THE ABOVE STEP. }

  IF (numComponents <> 2) and (GSet <> nil) THEN
    Iterate( SetComponents, numComponents, ON, ONSize,
      GSet, ntupleSize );

  { STABILIZE A BLOCK TO FIND AN INTRANSITIVE GROUP OF ORDER 2
    IN GROUP G }
}

```

```

-----
H := StabBlock( G, SetComponents, ON, ONSize, ntupleSize );
DispSP( SetPairs, ONSize );
DispSC( SetComponents, ONSize );
END; { ELSE }
BlkSys := H
END; { BlkSys }

*****
* Segment ststbseg (SetStab segment)
*
* This segment contains the routines which find the set-wise
* stabilizer of a set of ntuples M in 2-group G where G is known
* to set-wise stabilize the set of ntuples ON and ON contains M.
*****

SEGMENT ststbseg;

CONST
  hugesize = 10000;

TYPE
  { MERGE/FIND POINTERS USED TO FIND THE ORBITS OF THE NTUPLES
    STORED IN THE ARRAY ON }

  NORbitTypePtr = @NORbitType;
  MFNORbitTypePtr = @MFNORbitType;
  NORbitEltPtr = @NORbitEltType;
  NORbitType = array[ 1..hugesize ] of MFNORbitType;
  MFNORbitType = RECORD
    CASE TopNTuple: boolean OF
      false: (Parent: MFNORbitTypePtr );
      true: (Size: integer;
             NTupleList: NORbitEltPtr );
    END;
  NORbitEltType = RECORD
    NIndex: integer;
    NLink, NBLINK: NORbitEltPtr;
  END;
  BitPtr = @BitArray;
  BitArray = array[ 1..hugesize ] of boolean;

{ ROUTINES WHICH HAVE NOT BEEN INCLUDED IN THIS LISTING:

CopyG: create a copy of a list of generators for a group.
DispMFN: dispose of the merge/find data structure used to find
orbits of ntuples.

*****
*

```

```

* PROCEDURE FindMainverse
*
* This procedure is passed a permutation a and a set of ntuples
* M. It returns the set of ntuples M times a inverse in sorted
* order in array Mainverse.
*
*****}
PROCEDURE FindMainverse( VAR Mainverse: SetTypePtr;
                        VAR MaSize: integer;
                        a: PtrGenElt; M: SetTypePtr;
                        MSize: integer; ntupleSize: integer );

TYPE
{ LINKED LIST OF NTUPLES USED TO SORT THE NTUPLES EASILY WHILE
  APPLYING PERMUTATION a TO SET M. THE SORTED LIST IS THEN
  TRANSFERRED TO ARRAY Mainverse.
}

NEltPtr = @NEltType;
NEltType = RECORD
    N: ntuple;
    FLink: NEltPtr;
END;

VAR
i: integer;
iMap: ntuple;
j: integer;
NEltList, NElt: NEltPtr;

{*****}
* PROCEDURE FindElseInsert
*
* This procedure either finds ntuple p in the linked list of
* ntuples NEltList, or it is added in its correct position so that
* NEltList is sorted in ascending order.
*
*****}
PROCEDURE FindElseInsert( p: ntuple; VAR NEltList: NEltPtr;
                        VAR ListSize: integer; ntupleSize: integer);

VAR
prev, cur: NEltPtr;
i: integer;
differ: boolean;
found: boolean;

BEGIN
prev := nil;
cur := NEltList;
found := false;

-----
WHILE (cur <> nil) and (not found) DO BEGIN
    i := 1;
    differ := false;
    WHILE (i <= ntupleSize) and (not differ) DO
        IF p[i] = cur@.N[i] THEN
            i := i + 1
        ELSE
            differ := true;
        IF differ THEN
            IF p[i] > cur@.N[i] THEN BEGIN
                prev := cur;
                cur := cur@.FLink;
            END ELSE
                cur := nil
            ELSE
                found := true;
        END; { WHILE }
        IF not found THEN BEGIN
            new( cur );
            ListSize := ListSize + 1;
            cur@.N := p;
            IF prev = nil THEN BEGIN
                cur@.fLink := NEltList;
                NEltList := cur;
            END ELSE BEGIN
                cur@.fLink := Prev@.fLink;
                prev@.fLink := cur;
            END;
        END; { FindElseInsert }
    END; { FindMainverse }

    MaSize := 0;
    NEltList := nil;
    FOR i := 1 to MSize DO BEGIN
        FOR j := 1 to ntupleSize DO
            iMap[j] := a@.GenInverse@[ M@[ i, j ] ];
        SortNtuple( iMap, ntupleSize );
        FindElseInsert( iMap, NEltList, MaSize, ntupleSize );
    END; { FOR }
    Mainverse := ampxnew( MaSize*12 );
    FOR i := 1 to MaSize DO BEGIN
        Mainverse@[ i ] := NEltList@.N;
        NElt := NEltList;
        NEltList := NEltList@.fLink;
        dispose( NElt );
    END; { FOR }
END; { FindMainverse }

{*****}
* FUNCTION Stabntuple
*
*****}

```

```

*      This function finds and returns a list of generators for
*      the stabilizer of ntuple p in the group generated by the list
*      of generators G. G is transitive on the orbit of ntuples ON
*      which contains the ntuple p. The function finds Stab(p)
*      using Schreier's algorithm. The top group in the tower of
*      stabilizers is G and the distinguished point for that group
*      is not a point, but the ntuple p. The orbit for the top group
*      is therefore an orbit of ntuples. For all groups under G
*      in the tower, including Stab(p), the distinguished point is
*      a point and the orbit is an orbit of points. Thus the top
*      group is not stored using a Group structure, but all groups
*      beneath it are stored using a Group structure.
*
*      VARIABLE DICTIONARY
*
*      Stabp: a pointer to the tower of groups under group G (i.e.
*      from Stab(p) down to the identity group.
*      - built using the regular Schreier's algorithm routines.
*      pOrbit: an array of pointers to transversal elements
*      corresponding to the ntuples in the orbit of p
*      - the orbit associated with group G
*      - if pOrbit[i] is nil then the ntuple at index i in
*      orbit ON has not been added to the (partial) orbit of
*      p yet
*      - if pOrbit[i] is not nil, then it points to the
*      transversal element associated with the ntuple at index
*      i in orbit ON.
*      pTranStart, pTranEnd: pointers to the start and end of a
*      queue (linked list) of transversal elements associated
*      with the (partial) orbit of ntuple p
*      base: the base used to create the tower of stabilizers
*      pointed at by Stabp (which is created by the regular
*      Schreier routines)
*      q: used to step through the generators in G one by one
*      expanding the tower of stabilizers pointed at by Stabp
*      r: used to step through the generators in G added before
*      q and apply them to new ntuples added to the orbit of
*      p by q
*      b: used to step through the elements in the queue of
*      transversal elements associated with the orbit of p, or
*      to step through the newly added transversal elements.
*      bMap: the ntuple that the ntuple associated with transversal
*      element b is mapped to
*      bposn: the index of ntuple bMap in orbit ON.
*
*****]

```

```

FUNCTION Stabntuple( G: PtrGenElt; p: ntuple; ntupleSize: integer;
ON: SetTypePtr; ONSize: integer ): PtrGenElt;

```

TYPE

```

{ array of pointers to transversal elements which are associated
with the ntuples in the orbit of p }

```

```

NtupleOrbit = @NtupleOrbitType;
NOElt = @NOEltType;
NtupleOrbitType = array[ 1..hugeSize ] of NOElt;
NOEltType = RECORD
    NOposn: integer;
    NOTran, NOTranInverse: permutation;
    NextNOElt: NOElt
END;

```

VAR

```

Stabp: Group;
pOrbit: NtupleOrbit;
b: NOElt;
q, r: PtrGenElt;
base: BaseArray;
baseIndex: integer;
pTranStart, pTranEnd, holdend: NOElt;
bMap: ntuple;
bposn: integer;
i: integer;
h1, h2: permutation;

```

REF

```

ZSize: integer;

```

```

{*****}
*      PROCEDURE AddpOrbit
*
*      This procedure adds a new transversal element to the
*      transversal for Stab(p) in G. The ntuple associated with
*      the new transversal element is at index qposn in orbit ON.
*      The transversal element of the ntuple is given by the product
*      of permutations h and g. The new transversal structure is
*      also linked on to the end of the transversal queue.
*
*****}

```

```

PROCEDURE AddpOrbit( pOrbit: NtupleOrbit; qposn: integer;
h, g: permutation; VAR pTranEnd: NOElt );

```

VAR

```

qNOElt: NOElt;

```

BEGIN

```

new( qNOElt );
qNOElt.NOposn := qposn;
qNOElt.NextNOElt := nil;
pTranEnd.NextNOElt := qNOElt;
pTranEnd := qNOElt;
qNOElt.NOTran := multperms( h, g, ZSize );
qNOElt.NOTranInverse := Invertperm( qNOElt.NOTran, ZSize );
pOrbit[qposn] := qNOElt;

```

```

END; { AddpOrbit }

BEGIN { Stabntuple }
  Stabp := IGroup( 1, ZSize );
  pOrbit := ampnew( ONSize*4 );
  FOR i := 1 TO ONSize DO
    pOrbit[i] := nil;
  base := ampnew( ZSize*4 );

  { create transversal element associated with ntuple p }

  new( b );
  b@.NOposn := SearchON( p, ON, ONSize, ntupleSize );
  b@.NextNOelt := nil;
  b@.NOTran := ampnew( ZSize*4 );
  b@.NOTranInverse := ampnew( ZSize*4 );
  FOR i := 1 TO ZSize DO BEGIN
    base[i] := i;
    b@.NOTran[i] := i;
    b@.NOTranInverse[i] := i;
  END;
  baseIndex := 2;
  pOrbit[ b@.NOposn ] := b;
  pTranStart := b;
  pTranEnd := b;

  { APPLY EACH GENERATOR IN G TO EVERY NTUPLE IN THE ORBIT OF P }

  q := G;
  WHILE q <> nil DO BEGIN
    holdend := pTranEnd;
    b := pTranStart;
    WHILE b <> holdend@.NextNOelt DO BEGIN

      { APPLY GENERATOR q TO NTUPLE ASSOCIATED WITH b }

      FOR i := 1 TO ntupleSize DO
        bMap[i] := q@.Gen@.ON[ b@.NOposn, i ];
      bposn := SearchON( bMap, ON, ONSize, ntupleSize );
      IF pOrbit[ bposn ] = nil THEN BEGIN

        { found a new ntuple in the orbit of ntuple p }

        AddpOrbit( pOrbit, bposn, b@.NOTran, q@.Gen, pTranEnd );
      END ELSE BEGIN

        { found chord between ntuples b and bmap }

        h1 := multperms( b@.NOTran, q@.Gen, ZSize );
        h2 := multperms( h1, pOrbit[ bposn ]@.NOTranInverse,
                        ZSize );
        dispose( h1 );
        IF not GroupElement( h2, Stabp ) THEN BEGIN
          AddGen( h2, Stabp, base, baseIndex );
        END; { IF }
        dispose( h2 );
      END; { ELSE }
      b := b@.NextNOelt;
    END; { WHILE }

    { APPLY ALL OTHER GENERATORS r TO THE NEW NTUPLES
      ADDED BY GENERATOR q }

    b := holdend@.NextNOelt;
    WHILE b <> nil DO BEGIN
      r := G;
      WHILE r <> q@.NextGenElt DO BEGIN
        FOR i := 1 TO ntupleSize DO
          bMap[i] := r@.Gen@.ON[ b@.NOposn, i ];
        bposn := SearchON( bMap, ON, ONSize, ntupleSize );
        IF pOrbit[ bposn ] = nil THEN BEGIN

          { FOUND A NEW NTUPLE IN THE ORBIT OF p }

          AddpOrbit( pOrbit, bposn, b@.NOTran, r@.Gen,
                    pTranEnd );
        END ELSE BEGIN

          { FOUND A CHORD BETWEEN NTUPLES b AND bMap }

          h1 := multperms( b@.NOTran, r@.Gen, ZSize );
          h2 := multperms( h1, pOrbit[ bposn ]@.NOTranInverse,
                          ZSize );
          dispose( h1 );
          IF not groupelement( h2, Stabp ) THEN BEGIN
            AddGen( h2, Stabp, base, baseIndex );
          END; { IF }
          dispose( h2 );
        END; { ELSE }
        r := r@.NextGenElt;
      END; { WHILE }
      b := b@.NextNOelt;
    END; { WHILE }
    q := q@.NextGenElt;
  END; { WHILE }

  { DISPOSE OF THE CREATED STRUCTURES }

  dispose( base );
  FOR i := 1 TO ONSize DO BEGIN
    dispose( pOrbit[i]@.NOTran );
    dispose( pOrbit[i]@.NOTranInverse );
    dispose( pOrbit[i] );
  END; { FOR }
  dispose( pOrbit );
  Stabntuple := Stabp@.Gens;
  Stabp@.Gens := nil;
  disposeTower( Stabp );

```

```

END; { Stabntuple }

{*****}
*
*   PROCEDURE CreateNTupleMap
*
*   Create the mapping from the index of an ntuple in the orbit
*   of ntuples, ON, to its merge/find pointer. The group has
*   changed so ON is no longer an orbit of ntuples, so the new orbits
*   within ON are found using these merge/find pointers.
*
*   {*****}

PROCEDURE CreateNTupleMap( VAR NTupleMap: NORbitTypePtr;
                           ONSize: integer );

VAR
  MFN: MFNORbitPtr;
  NE: NORbitEltPtr;
  i: integer;

BEGIN
  NTupleMap := ampxnew( ONSize*4 );
  FOR i := 1 to ONSize DO BEGIN
    new(MFN);
    NTupleMap@[ i ] := MFN;
    MFN@.TopNTuple := true;
    MFN@.Size := 1;
    new( NE );
    MFN@.NtupleList := NE;
    NE@.Nindex := i;
    NE@.NFLink := NE;
    NE@.NBLink := NE;
  END; { FOR }
END; { CreateNTupleMap }

{*****}
*
*   FUNCTION FNOorbit
*
*   This function returns a pointer to the root of the merge/find
*   tree which contains the merge/find element at index i in the
*   array NTupleMap. It also compresses the path from NTupleMap[i]
*   to the root by pointing all elements on the path at the root.
*
*   {*****}

FUNCTION FNOorbit( i: integer; NTupleMap: NORbitTypePtr ): MFNORbitPtr;

VAR
  MFi, MFj, MFk: MFNORbitPtr;

BEGIN
  MFi := NTupleMap@[ i ];

  WHILE not MFi@.TopNTuple DO
    MFi := MFi@.Parent;
  IF MFi <> NTupleMap@[ i ] THEN BEGIN
    MFj := NTupleMap@[ i ];
    WHILE MFj@.Parent <> MFi DO BEGIN
      MFk := MFj@.Parent;
      MFj@.Parent := MFi;
      MFj := MFk;
    END; { WHILE }
  END; { IF }
  FNOorbit := MFi;
END; { FNOorbit }

{*****}
*
*   PROCEDURE MergeNORbit
*
*   This procedure is passed pointers to the roots of two merge/
*   find trees used to find the orbits of ntuples in the array ON.
*   The procedure merges the two trees by pointing the root of the
*   smaller tree at the root of the larger tree and by amalgamating
*   the two circularly linked lists of elements in the trees.
*
*   {*****}

PROCEDURE MergeNORbit( MFi, MFj: MFNORbitPtr );

VAR
  MFk: MFNORbitPtr;
  NEi, NEj: NORbitEltPtr;

BEGIN
  IF MFi@.Size < MFj@.Size THEN BEGIN
    MFk := MFi;
    MFi := MFj;
    MFj := MFk;
  END;
  NEi := MFi@.NtupleList@.NBLink;
  NEj := MFj@.NtupleList@.NBLink;
  MFi@.NtupleList@.NBLink := NEj;
  NEj@.NFLink := MFi@.NtupleList;
  MFj@.NtupleList@.NBLink := NEi;
  NEi@.NFLink := MFj@.NtupleList;
  MFi@.Size := MFi@.Size + MFj@.Size;
  MFj@.TopNTuple := false;
  MFj@.Parent := MFi;
END; { MergeNORbit }

{*****}
*
*   PROCEDURE FNObits
*
*   This procedure uses a merge/find algorithm to find the orbits
*   of the unused ntuples in the array ON. If the ntuple at index i

```

```

* in ON is unused, then ONUsed@[ i ] will be false and the merge/
* find pointer for the ntuple will be at index i in NtupleMap.
*
*****]
PROCEDURE FNORbits( NtupleMap: NorbitTypePtr; ON: SetTypePtr;
  ONSize: integer; ONUsed: BitPtr;
  H: PtrGenElt; ntupleSize: integer );

VAR
  r: PtrGenElt;
  i,j,k: integer;
  MFi, MFj: MPNOorbitPtr;
  iMap: ntuple;

REF
  ZSize: integer;

BEGIN
  FOR i := 1 to ONSize DO
    IF not ONUsed@[ i ] THEN BEGIN
      r := H;
      WHILE r <> nil DO BEGIN
        FOR k := 1 to ntupleSize DO
          iMap[ k ] := r.Gen@[ ON@[ i, k ] ];
        j := SearchON( iMap, ON, ONSize, ntupleSize );
        MFi := FNORbit( i, NtupleMap );
        MFj := FNORbit( j, NtupleMap );
        IF MFi <> MFj THEN BEGIN
          MergeNOorbit( MFi, MFj );
        END;
        r := r.NextGenElt;
      END; { WHILE }
    END; { IF }
  END; { FNORbits }

  { *****
  *
  *   PROCEDURE CreateONi
  *
  *   NEONi points at a circularly linked list containing the
  *   indices of ntuples in the array ON. This procedure creates a
  *   new array, ONi, and puts the ntuples in ONi in sorted order.
  *   A heapsort of the indices is used to sort the ntuples.
  *
  *   *****
  *
  *   PROCEDURE CreateONi ( VAR ONi: SetTypePtr; ONiSize: integer;
  *     NEONi: NorbitEltPtr; ntupleSize: integer;
  *     ON: SetTypePtr; ONUsed: BitPtr );

  VAR
    i,j: integer;
    NE: NorbitEltPtr;

```

```

temp: integer;

{ *****
*
*   PROCEDURE HeapUpOni
*
*   ONi contains indices of ntuples in array ON in the first
*   position in rows 1 to heapend. Rows 1 to heapend-1 form a heap.
*   The new element in position heapend must be "bubbled" up to its
*   correct position, i.e. until it is smaller than its parent.
*
*   *****
*
*   PROCEDURE HeapUpOni( ONi: SetTypePtr; heapend: integer );

  VAR
    newelt, parent: integer;
    temp: integer;
    i: integer;

  BEGIN
    newelt := heapend;
    parent := newelt div 2;
    temp := ONi@[ newelt, 1 ];
    WHILE parent <> 0 DO
      IF ONi@[ parent, 1 ] >= temp THEN
        parent := 0
      ELSE BEGIN
        ONi@[ newelt, 1 ] := ONi@[ parent, 1 ];
        ONi@[ parent, 1 ] := temp;
        newelt := parent;
        parent := newelt div 2;
      END; { ELSE }
    END; { HeapUpOni }

  { *****
  *
  *   PROCEDURE HeapDownOni
  *
  *   This procedure is passed array ONi which contains a heap in
  *   position 1 of rows 1 to heapend, except that the first row is
  *   out of place. It must be "bubbled" down to the correct position
  *   i.e. where it is smaller than its parent but larger than its
  *   children (if it has any).
  *
  *   *****
  *
  *   PROCEDURE HeapDownOni( ONi: SetTypePtr; heapend: integer );

  VAR
    newelt, child1, child2: integer;
    temp: integer;
    largest: integer;
    i,j: integer;

```

```

BEGIN
  newelt := 1;
  temp := ONi@[ 1, 1 ];
  WHILE newelt <> 0 DO BEGIN
    child1 := newelt*2;
    child2 := child1 + 1;
    largest := 0;
    IF child2 <= heapend THEN BEGIN
      IF temp < ONi@[ child1, 1 ] THEN
        IF ONi@[ child1, 1 ] > ONi@[ child2, 1 ] THEN
          largest := child1
        ELSE
          largest := child2
      ELSE
        IF temp < ONi@[ child2, 1 ] THEN
          largest := child2
      END ELSE
        IF child1 = heapend THEN
          IF temp < ONi@[ child1, 1 ] THEN
            largest := child1;
          IF largest = 0 THEN
            newelt := 0
          ELSE BEGIN
            ONi@[ newelt, 1 ] := ONi@[ largest, 1 ];
            ONi@[ largest, 1 ] := temp;
            newelt := largest
          END; { ELSE }
        END; { WHILE }
      END; { HeapDownONi }
    { MAINLINE OF CreateONi }
  BEGIN
    ONi := ampxnew( ONiSize*12 );
    NE := NEONi@.NFLink;
    ONi@[ 1, 1 ] := NEONi@.Nindex;
    ONUsed@[ NEONi@.Nindex ] := true;
    FOR i := 2 to ONiSize DO BEGIN
      ONi@[ i, 1 ] := NE@.Nindex;
      ONUsed@[ NE@.Nindex ] := true;
      HeapUpONi( ONi, i );
      Ne := Ne@.NFLink;
    END; { FOR }
    FOR i := ONiSize DOWNT0 2 DO BEGIN
      temp := ONi@[ i, 1 ];
      ONi@[ i, 1 ] := ONi@[ 1, 1 ];
      ONi@[ 1, 1 ] := temp;
      HeapDownONi( ONi, i-1 );
    END; { FOR }
    FOR i := 1 to ONiSize DO BEGIN
      temp := ONi@[ i, 1 ];
      FOR j := 1 to ntuplesize DO
        ONi@[ i, j ] := ON@[ temp, j ];
    END; { FOR }
  END; { CreateONi }

  {*****}
  *
  *   PROCEDURE CreateMi
  *
  *   This procedure is passed an orbit of ntuples in array ONi.
  *   It returns the intersection of the ntuples in ONi with the
  *   ntuples in array M in a new array Mi.
  *
  *   {*****}
  PROCEDURE CreateMi( VAR Mi: SetTypePtr; VAR MiSize: integer;
    MUsed: BitPtr; M: SetTypePtr; MSize: integer;
    VAR MLeft: integer; ONi: SetTypePtr;
    ONiSize: integer; ntuplesize: integer );

  VAR
    i,j: integer;
    iposn: integer;
    MiEltList, MiElt: NORbitEltPtr;
    iconsidered: integer;

  REF
    output: text;

  BEGIN
    MiEltList := nil;
    MiSize := 0;
    IF MLeft <= ONiSize THEN BEGIN
      i := 1;
      iconsidered := 0;
      WHILE iconsidered < MLeft DO BEGIN
        IF not MUsed[ i ] THEN BEGIN
          iconsidered := iconsidered + 1;
          iposn := SearchON( M@[ i ], ONi, ONiSize, ntuplesize );
          IF iposn <> 0 THEN BEGIN
            MiSize := MiSize + 1;
            new( MiElt );
            MiElt@.Nindex := i;
            IF MiEltList = nil THEN BEGIN
              MiElt@.NFLink := MiElt;
              MiElt@.NBLink := MiElt;
              MiEltList := MiElt
            END ELSE BEGIN
              MiElt@.NFLink := MiEltList;
              MiElt@.NBLink := MiEltList@.NBLink;
              MiEltList@.NBLink@.NFLink := MiElt;
              MiEltList@.NBLink := MiElt;
            END;
          END;
          iconsidered := i;
        END; { IF }
        i := i + 1;
      END;
    END;
  END;

```



```

END; { WHILE }
END ELSE BEGIN
  FOR i := 1 to ONiSize DO BEGIN
    iposn := SearchON(ONi@[ i ], M, MSize, ntupleSize );
    IF iposn <> 0 THEN BEGIN
      MiSize := MiSize + 1;
      new( MiElt );
      MiElt@.NIndex := iposn;
      IF MiEltList = nil THEN BEGIN
        MiElt@.NFLink := MiElt;
        MiElt@.NBLink := MiElt;
        MiEltList := MiElt;
      END ELSE BEGIN
        MiElt@.NFLink := MiEltList;
        MiElt@.NBLink := MiEltList@.NBLink;
        MiEltList@.NBLink@.NFLink := MiElt;
        MiEltList@.NBLink := MiElt;
      END;
    END; { IF }
  END; { FOR }
END; { ELSE }
CreateONi( Mi, MiSize, MiEltList, ntupleSize, M, MUsed );
MLeft := MLeft - MiSize;
MiEltList@.NBLink@.NFLink := nil;
WHILE MiEltList <> nil DO BEGIN
  MiElt := MiEltList@.NFLink;
  dispose( MiEltList );
  MiEltList := MiElt;
END;
END; { CreateMi }

{*****}
*
*   PROCEDURE ReOrderMFN
*
*   This procedure reorders the merge/find pointers of the ntuples*
*   in array ONi that have not been used yet. Each is set to be in *
*   its own orbit of ntuples containing only itself, so that the *
*   new orbits can be found.
*
*   {*****}

PROCEDURE ReOrderMFN( NTupleMap: NORbitTypePtr; NSize: integer;
  ONUsed: BitPtr );

VAR
  i: integer;
  NE1, NE2: NORbitEltPtr;

BEGIN
  FOR i := 1 to NSize DO
    IF (not ONUsed@[i]) and (NTupleMap@[i]@.TopNTuple) THEN
      IF NTupleMap@[i]@.Size > 1 THEN BEGIN
        NE1 := NTupleMap@[ i ]@.NTupleList;

        NE1@.NBLink@.NFLink := nil;
        WHILE NE1 <> nil DO BEGIN
          NE2 := NE1@.NFLink;
          NE1@.NFLink := NE1;
          NE1@.NBLink := NE1;
          NTupleMap@[ NE1@.NIndex ]@.TopNTuple := true;
          NTupleMap@[ NE1@.NIndex ]@.Size := 1;
          NTupleMap@[ NE1@.NIndex ]@.NTupleList := NE1;
          NE1 := NE2;
        END; { WHILE }
      END; { IF }
    END; { ReOrderMFN }

    {*****}
    *
    *   FUNCTION SetStab
    *
    *   This function is passed a 2-group G, a set of ntuples ON
    *   which G setwise stabilizes, and a subset M of ON. The function
    *   finds the setwise stabilizer of M in G using a divide-and-
    *   conquer technique: G is intransitive on the set ON, so ON can
    *   be broken down into orbits ONi. Let Mi be the intersection of
    *   ONi and M. Then G must setwise stabilize Mi. G is transitive
    *   on ONi so find Stab(Mi) in G by calling TranSetStab. Repeat the
    *   procedure on Stab(Mi) using an orbit ONi' of Stab(Mi) and Mi',
    *   the intersection of ONi' and M.
    *
    *   {*****}

    FUNCTION SetStab( G: PtrGenElt; ntupleSize: integer;
      ON: SetTypePtr; ONSize: integer;
      M: SetTypePtr; MSize: integer): PtrGenElt; EXTERNAL;

    VAR
      ONUsed: BitPtr; { ONUsed@[i] = true => ON@[i] has
        been deleted }
      MUsed: BitPtr; { MUsed@[i] = true => M@[i] has
        been deleted }
      Mi: SetTypePtr; { the intersection of M and ONi }
      MLeft: integer; { # of ntuples left undeleted in M }
      MUnused: integer; { 1st ntuple in M that has not been
        used so far }

      MiSize: integer;
      HP, H: PtrGenElt;
      ONi: SetTypePtr; { an orbit of ntuples of G which
        is a subset of ON }

      ONiSize: integer;
      NTupleMap: NORbitTypePtr; { maps ntuples in ON to their merge/
        find pointers }

      MFONi: MFNORbitPtr;
      i: integer;
      NE: NORbitEltPtr;

```

```

REF
ZSize: integer;

BEGIN
  ONUsed := ampnew( ONSize );
  FOR i := 1 to ONSize DO
    ONUsed[i] := false;
  MUsed := ampnew( MSize );
  FOR i := 1 to MSize DO
    MUsed[i] := false;
  MLeft := MSize;
  MUnused := 1;
  H := CopyG( G );
  HP := H;
  CreateNTupleMap( NTupleMap, ONSize );

  { CONTINUE FINDING ORBITS ONi AND SUBSETS Mi UNTIL THE ENTIRE SET M
    IS SET-WISE STABILIZED. }

  WHILE (MLeft > 0) AND (H <> nil) DO BEGIN
    { FIND THE ORBITS OF THE NTUPLES IN SET ON WHICH HAVE NOT YET
      BEEN USED. }

    FNorbits( NTupleMap, ON, ONSize, ONUsed, H, ntupleSize );
    WHILE MUsed[ MUnused ] DO MUnused := MUnused + 1;
    MFONi := FNorbit( SearchON( M[ MUnused ], ON, ONSize, NtupleSize ),
      NTupleMap );
    ONiSize := MFONi.Size;
    CreateONi( ONi, ONiSize, MFONi.NtupleList, ntupleSize, ON,
      ONUsed );
    CreateMi( Mi, MiSize, MUsed, M, MSize, MLeft, ONi, ONiSize,
      ntupleSize );

    { IF Mi AND ONi ARE THE SAME SIZE THEN Mi IS ALREADY SET-WISE
      STABILIZED SO WE DO NOT HAVE TO DO ANYTHING.
      IF Mi CONTAINS ONE NTUPLE THEN WE CAN FIND THE SET-WISE
      STABILIZER OF Mi USING SCHREIER'S ALGORITHM.
      OTHERWISE WE MUST CALL TranSetStab TO FIND STAB(Mi). }

    IF MiSize <> ONiSize THEN BEGIN
      IF MiSize < 1 THEN BEGIN
        HP := TranSetStab( H, ntupleSize, ONi, ONiSize, Mi, MiSize );
        DispGenList( H );
        H := HP;
      END ELSE BEGIN
        HP := Stabntuple( H, Mi[ 1 ], ntupleSize, ONi, ONiSize );
        DispGenList( H );
      END; { ELSE }
    END; { IF }
    dispose( ONi );
    dispose( Mi );
    ReOrderMFN( NTupleMap, ONSize, ONUsed );
  END;

```

```

END; { WHILE }
DispMFN( NTupleMap, ONSize );
dispose( ONUsed );
dispose( MUsed );
SetStab := H;
END; { SetStab }

```

```

{*****}
*
* FUNCTION TranCosetStab
*
* G is a 2-group acting transitively on the orbit of ntuples
* ON. M and N are both subsets of ON. This function searches the
* group G for a permutation Q which maps the set of ntuples M to
* the set of ntuples N.
* First a subgroup P of G is found. P is a normal subgroup of
* G fixing a block system for G containing 2 blocks. Thus, P has
* index 2 in G.
*
* First, the group P is searched for a permutation Q mapping
* M to N. If there is such a Q, then return it.
* If P did not contain such a Q, then if Stab(M) in G is
* the subgroup P of P, then coset Pc could contain such a Q.
*
* Thus, we should first check if Stab(M) = P, and then, if it
* is, check coset Pc for such a Q. Since there are only two cosets
* of P in G, we skip the check if Stab(M) = P and just search
* P and Pc for the required Q.
*
* {*****}

```

```

FUNCTION TranCosetStab( G: PtrGenElt; N: SetTypePtr; NSize: integer;
  ntupleSize: integer;
  ON: SetTypePtr; ONSize: integer;
  M: SetTypePtr; MSize: integer); PtrGenElt;
EXTERNAL;

```

```

VAR
P: Group; { An subgroup of G fixing a block
            system of 2 blocks for G }
Q: PtrGenElt;
C: PtrGenElt; { a permutation in G that is not in P }
temp: permutation;
i, j: integer;
Ni: SetTypePtr; { (N)C**-1 }
NiSize: integer;

```

```

REF
ZSize: integer;

```

```

BEGIN
{ FIND AN INTRANSITIVE NORMAL SUBGROUP P OF G OF INDEX 2. G IS
IMPRIMITIVE. P WILL FIX A BLOCK SYSTEM. }

P := BlkSys( G, ntupleSize, ON, ONSize );
{ P MAY CONTAIN THE DESIRED PERMUTATION Q }
Q := CosetStab( P@.Gens, N, NSize, ntupleSize, ON, ONSize,
M, MSize );

{ IF P DOES CONTAIN SUCH A Q, RETURN IT. OTHERWISE, SOME COSET
OF P IN G MAY CONTAIN THE DESIRED Q }

IF Q = nil THEN BEGIN
{ FIND A PERMUTATION C NOT IN P TO REPRESENT COSET P C }

C := G;
WHILE GroupElement( C@.Gen, P ) DO
C := C@.NextGenElt;

{ CHECK COSET P C FOR Q MAPPING M TO N }

FindMaInverse( Ni, NiSize, C, N, NSize, ntupleSize );
Q := CosetStab( P@.Gens, Ni, NiSize, ntupleSize,
ON, ONSize, M, MSize );
dispose( Ni );
IF Q <> nil THEN BEGIN
temp := MultPerms( Q@.Gen, C@.Gen, ZSize );
dispose( Q@.Gen );
Q@.Gen := temp;
dispose( Q@.GenInverse );
Q@.GenInverse := InvertPerm( Q@.Gen, ZSize );
END;
{ IF }
DisposeTower( P );
TranCosetStab := Q;
END; { TranCosetStab }

{*****}
*
* PROCEDURE FindNi
*
* This procedure is passed two sets of ntuples, N and ONi, and
* a permutation Q. This procedure finds the intersection of ONi
* and (N multiplied by Q**-1) and returns this in set Ni. The set
* Ni must be ordered so that it can be searched easily. Since ONi
* is already ordered, the ordering of Ni is done by looking at
* the index of each ntuple in ONi.
*
* {*****}

```

```

PROCEDURE FindNi( VAR Ni: SetTypePtr; VAR NiSize: integer;
N: SetTypePtr; NSize: integer;
ON: SetTypePtr; ONSize: integer;
Q: PtrGenElt; ntupleSize: integer );

```

```

TYPE
NEltPtr = @NEltType;
NEltType = RECORD
NIndex: integer;
Nflink: NEltPtr;
END;

```

```

VAR
iMap: ntuple;
iposn: integer;
i,j: integer;
NEltList, NElt: NEltPtr;

```

```

{*****}
*
* PROCEDURE FindElseInsert
*
* This procedure finds the ntuple at position iposn in the
* set of ntuples ONi in the linked list of NEltType nodes pointed
* at by NEltList, or it inserts iposn into the list in its correct
* position in ascending order.
*
* {*****}

```

```

PROCEDURE FindElseInsert( VAR NEltList: NEltPtr;
iposn: integer; VAR NSize: integer );

```

```

VAR
prev, cur: NEltPtr;
found: boolean;

```

```

BEGIN
prev := nil;
cur := NEltList;
found := false;
WHILE (cur <> nil) and (not found) DO
IF iposn = cur@.NIndex THEN
found := true;
ELSE
IF iposn > cur@.NIndex THEN BEGIN
prev := cur;
cur := cur@.Nflink;
END ELSE
cur := nil;
IF not found THEN BEGIN
new( cur );
cur@.NIndex := iposn;
NSize := NSize + 1;
IF prev = nil THEN BEGIN

```

```

        cur@.Nflink := NeltList;
        NeltList := cur;
    END ELSE BEGIN
        cur@.Nflink := prev@.nflink;
        prev@.Nflink := cur;
    END; { ELSE }
END; { IF }
END; { FindElseInsert }

BEGIN { FindNi }
    NiSize := 0;
    NeltList := nil;
    FOR i := 1 to NiSize DO BEGIN
        FOR j := 1 to ntupleSize DO
            iMap[ j ] := Q@.GenInverse@[ N@[ i, j ] ];
            iposn := SearchON( iMap, ON, ONSize, ntupleSize );
            IF iposn <> 0 THEN
                FindElseInsert( NeltList, iposn, NiSize );
            END;
        END;
        Ni := ampnew( NiSize*12 );
        FOR i := 1 to NiSize DO BEGIN
            Nelt := NeltList@.Nflink;
            Ni@[ i ] := ON@[ NeltList@.Nindex ];
            dispose( NeltList );
            NeltList := Nelt;
        END; { FOR }
    END; { FindNi }
END;

```

```

{*****}
*
* FUNCTION SearchCosetDiag
*
* M and n are ntuples. They are both in the orbit of ntuples
* ONi. This function searches the coset diagram of Stab( m ) in
* H to find a permutation which maps m to n. If there is no such
* permutation, nil is returned.
*
{*****}

```

```

FUNCTION SearchCosetDiag( m,n: ntuple; ONi: SetTypePtr;
    ONiSize: integer; ntupleSize: integer;
    H: PtrGenElt ): PtrGenElt;

```

TYPE

```

{ A STRUCTURE REPRESENTING THE ORBIT OF NTUPLE m }

```

```

mPtr = @MType;
mOEltPtr = @mOElt;
MType = array[ 1..hugesize ] of mOEltPtr;
mOElt = RECORD
    posn: integer;
    perm: permutation;

```

```

        NextmOElt: mOEltPtr;
    END;

```

VAR

```

mOrbit: mPtr;
TranStart, TranEnd: mOEltPtr;
T1, T2: mOEltPtr;
i, j: integer;
nposn: integer;
T1Map: ntuple;
Q1, Q2: PtrGenElt;
HoldEnd: mOEltPtr;

```

REF

```

ZSize: integer;

```

BEGIN

```

{ CREATE THE STRUCTURE mOrbit REPRESENTING THE ORBIT OF m,
  AND INITIALIZE IT TO CONTAIN ONLY m (WITH TRANSVERSAL
  ELEMENT EQUAL TO THE IDENTITY). }

```

```

new(T1);
T1@.posn := SearchON( m, ONi, ONiSize, ntupleSize );
T1@.perm := ampnew( ZSize*4 );
FOR i := 1 to ZSize DO
    T1@.perm@[ i ] := i;
T1@.NextmOElt := nil;
mOrbit := ampnew( ONiSize*4 );
FOR i := 1 to ONiSize DO
    mOrbit@[ i ] := nil;
mOrbit@[ T1@.posn ] := T1;
TranStart := T1;
TranEnd := T1;
nposn := SearchON( n, ONi, ONiSize, ntupleSize );
Q1 := H;

```

```

{ APPLY THE GENERATORS OF GROUP H TO THE ELEMENTS IN THE ORBIT
  OF m UNTIL WE HAVE FOUND THE TRANSVERSAL ELEMENT MAPPING
  m TO n. }

```

```

WHILE ( Q1 <> nil) and ( mOrbit@[ nposn ] = nil) DO BEGIN
    { APPLY GENERATOR Q1 TO EVERYTHING IN THE ORBIT OF NTUPLE
      M SO FAR

```

```

    T1 := TranStart;
    HoldEnd := TranEnd;
    WHILE ( T1 <> HoldEnd@.NextmOElt) and (mOrbit@[ nposn ] = nil) DO
        BEGIN
            FOR i := 1 to ntupleSize DO BEGIN
                T1Map[ i ] := Q1@.Gen@[ ONi@[ T1@.posn, i ] ];
            END;
            j := SearchON( T1Map, ONi, ONiSize, ntupleSize );
            IF mOrbit@[ j ] = nil THEN BEGIN

```

```

{ GENERATOR Q1 MAPPED NTUPLE T1 TO AN NTUPLE WHICH
  IS NOT CURRENTLY IN THE ORBIT OF m, SO ADD THE
  NEW NTUPLE (AND ITS TRANSVERSAL ELEMENT) TO THE
  ORBIT.
}

new( T2 );
mOrbit[ j ] := T2;
T2@.posn := j;
T2@.perm := MultPerms( T1@.perm, Q1@.Gen, ZSize );
T2@.NextmOElt := nil;
TranEnd@.NextmOElt := T2;
TranEnd := T2;
END; { IF }
T1 := T1@.NextmOElt;
END; { WHILE }
HoldEnd := HoldEnd@.NextmOElt;

{ APPLY ALL GENERATORS WHICH HAVE BEEN PROCESSED TO THE
  NEWLY ADDED NTUPLES IN THE ORBIT OF m.
}

WHILE (HoldEnd <> nil) and (mOrbit[ nposn ] = nil) DO BEGIN
  Q2 := H;
  WHILE (Q2 <> Q1@.NextGenElt) and (mOrbit[ nposn ] = nil)
  DO BEGIN
    FOR i := 1 to ntupleSize DO
      T1Map[ i ] := Q2@.Gen[ ONi[ HoldEnd@.posn, i ] ];
    j := SearchON( T1Map, ONi, ONiSize, ntupleSize );
    IF mOrbit[ j ] = nil THEN BEGIN
      { GENERATOR Q2 MAPPED NTUPLE T1 TO AN NTUPLE
        WHICH IS NOT CURRENTLY IN THE ORBIT, SO ADD IT.
      }

      new( T2 );
      mOrbit[ j ] := T2;
      T2@.posn := j;
      T2@.perm := MultPerms( HoldEnd@.perm, Q2@.Gen, ZSize );
      T2@.NextmOElt := nil;
      TranEnd@.NextmOElt := T2;
      TranEnd := T2;
      END; { IF }
      Q2 := Q2@.NextGenElt;
    END; { WHILE }
    HoldEnd := HoldEnd@.NextmOElt;
  END; { WHILE }
  Q1 := Q1@.NextGenElt;
END; { WHILE }
IF mOrbit[ nposn ] <> nil THEN BEGIN
  new( Q1 );
  Q1@.Gen := mOrbit[ nposn ]@.perm;
  Q1@.GenInverse := InvertPerm( Q1@.Gen, ZSize );
  Q1@.NextGenElt := nil;
  mOrbit[ nposn ]@.perm := nil;
  dispose( mOrbit[ nposn ] );

```

```

END ELSE
  Q1 := nil;
FOR i := 1 to ONiSize DO
  IF mOrbit[ i ] <> nil THEN BEGIN
    dispose( mOrbit[ i ]@.perm );
    dispose( mOrbit[ i ] );
  END;
dispose( mOrbit );
SearchCosetDiag := Q1;
END; { SearchCosetDiag }

```

```

*****
* FUNCTION CosetStab
*
* P is an intransitive 2-group acting on the orbit of ntuples
* ON. We search for an element Q which maps the set of ntuples
* M to the set of ntuples N, where Q is in group P. (M and N are
* subsets of the set ON.)
* If such a Q exists, it is returned. Otherwise, a nil pointer
* is returned.
* The procedure used here is a divide-and-conquer technique
* similar to that used in FUNCTION SetStab. We find the orbits
* of ntuples within ON. Let ONi be an orbit of ntuples within
* ON. Find Mi and Ni, the intersections of M and N with ONi. If
* there is a Q mapping M to N, it must map Mi to Ni. So we search
* for Qi mapping Mi to Ni. Now we must find a Q mapping the
* rest of M to the rest of N, i.e. mapping M-Mi to N-Ni. We no
* search Stab(Mi) for a Q mapping M-Mi to (N-Ni)Qi**-1, using the
* same technique that we used to find Qi.
* If, at any point in the procedure, we find that there is no
* such Qi mapping Mi to Ni then we know that there is no such Q
* mapping M to N. In this case, we stop and return nil. Thus, if
* Mi is not the same size as Ni, we can stop and return nil.
*****

```

```

FUNCTION CosetStab( P: PtrGenElt; N: SetTypePtr; NSize: integer;
  ntupleSize: integer;
  ON: SetTypePtr; ONSize: integer;
  M: SetTypePtr; MSize: integer ): PtrGenElt;
EXTERNAL;

```

```

VAR
  HP, H: PtrGenElt;
  ONUsed: BitPtr; { tells us what parts of ON have
                   already been processed. }
  MUsed: BitPtr; { tells us what parts of M have
                  already been used. }
  Mi: SetTypePtr; { M intersected with orbit ONi }
  MiSize: integer;
  MLeft: integer; { # of ntuples left unused in M }
  MUnused: integer; { 1st ntuple in M which is unused }

```

```

Oni: SetTypePtr;           { an orbit of ntuples in ON }
OniSize: integer;
NtupleMap: NorbitTypePtr;  { mapping from the index of an ntuple
                             in ON to its merge/find pointer;
                             Used to find orbits of ntuples }
MFONi: MFNorbitPtr;        { pointer to the root of the merge/
                             find tree representing orbit Oni }

Q: PtrGenElt;
i: integer;
Ni: SetTypePtr;           { ONi intersected with (N)Q*-1 }
NiSize: integer;
Qi: PtrGenElt;
temp: NorbitEltPtr;

REF
ZSize: integer;

BEGIN
  OnUsed := ampxnew( ONSize );
  FOR i := 1 TO ONSize DO
    ONUsed[i] := false;
  MUsed := ampxnew( MSize );
  FOR i := 1 TO MSize DO
    MUsed[i] := false;
  MLeft := MSize;
  MUnused := 1;
  H := CopyG( P );
  new( Q );
  Q@.NextGenElt := nil;
  Q@.Gen := ampxnew( ZSize*4 );
  Q@.GenInverse := ampxnew( ZSize*4 );
  FOR i := 1 TO ZSize DO BEGIN
    Q@.Gen[i] := i;
    Q@.GenInverse[i] := i;
  END; { FOR }
  CreateNtupleMap( ntupleMap, ONSize );

  { CONTINUE THE DIVIDE-AND-CONQUER TECHNIQUE UNTIL
    EITHER WE KNOW THAT NO SUCH Q EXISTS OR UNTIL ALL
    OF M IS USED UP }

  WHILE ( MLeft > 0 ) and ( H <> nil ) DO BEGIN
    { FIND THE ORBITS OF THE NTUPLES IN ON WHICH HAVE NOT BEEN
      PREVIOUSLY USED. }

    FNOrbits( NtupleMap, ON, ONSize, ONUsed, H, NtupleSize );

    { CHOOSE AN ORBIT WHICH CONTAINS AN UNUSED ELEMENT OF M }

    WHILE MUsed[MUnused] DO MUnused := MUnused + 1;
    MFONi := FNOrbit( SearchON( M[MUnused], ON, ONSize, ntupleSize),
                     ntupleMap );
    OniSize := MFONi@.Size;

```

```

  CreateOni( Oni, OniSize, MFONi@.ntupleList, ntupleSize,
             ON, ONUsed );
  CreateMi( Mi, MiSize, MUsed, M, MSize, MLeft, Oni, OniSize,
            ntupleSize );
  FindNi( Ni, NiSize, N, NSize, Oni, OniSize, Q, ntupleSize );

  { WE WANT Qi IN COSET H Q THAT MAPS Mi TO Ni }

  IF NiSize <> MiSize THEN BEGIN
    DispGenList( H );
    dispose( Q@.Gen );
    dispose( Q@.GenInverse );
    dispose( Q );
    Q := nil;
  END ELSE IF MiSize <> OniSize THEN BEGIN
    { IF Mi = Ni = Oni THEN Q ALREADY MAPS Mi TO Ni
      AND H ALREADY SET-WISE STABILIZES Mi. }

    IF MiSize = 1 THEN BEGIN
      Qi := SearchCosetDiag( Mi@[ 1 ], Ni@[1], Oni, OniSize,
                           ntupleSize, H );
    END ELSE BEGIN
      Qi := TranCosetStab( H, Ni, NiSize, ntupleSize,
                          Oni, OniSize, Mi, MiSize );
    END; { ELSE }
    IF Qi = nil THEN BEGIN
      DispGenList( H );
      dispose( Q@.Gen );
      dispose( Q@.GenInverse );
      dispose( Q );
      Q := nil;
    END ELSE BEGIN
      { Qi MAPS Mi TO Ni. THE SET OF ALL SUCH ELEMENTS IS A
        COSET OF THE SETWISE STABILIZER OF Mi IN H }

      IF MLeft > 0 THEN BEGIN
        IF MiSize <> 1 THEN BEGIN
          HP := TranSetStab( H, ntupleSize, Oni, OniSize, Mi,
                           MiSize );
        END ELSE BEGIN
          HP := Stabntuple( H, Mi@[ 1 ], ntupleSize,
                          Oni, OniSize );
        END; { ELSE }
        DispGenList( H );
        H := HP;
      END; { IF }
      new( HP );
      HP@.Gen := MultPerms( Qi@.Gen, Q@.Gen, ZSize );
      dispose( Q@.Gen );
      Q@.Gen := HP@.Gen;
      dispose( HP );
      dispose( Q@.GenInverse );

```

```

        Q@.GenInverse := InvertPerm( Q@.Gen, ZSize );
        dispose( Q1@.Gen );
        dispose( Q1@.GenInverse );
        dispose( Q1 );
    END; { ELSE }
END; { ELSE }
dispose( ON1 );
dispose( M1 );
dispose( N1 );
ReOrderMFN( NtupleMap, ONSize, ONUsed );
END; { WHILE }
DispMFN( NtupleMap, ONSize );
dispose( ONUsed );
dispose( MUsed );
IF H <> nil THEN
    DispGenList( H );
CosetStab := Q;
END; { CosetStab }

{*****}
*
* FUNCTION TranSetStab
*
* We want to find the set-wise stabilizer of the set of ntuples
* M in group G. G is transitive on the orbit of ntuples ON
* which contains the set of ntuples M. Stab(M) is found by finding
* an intransitive group P of index 2 in G (using a block system).
* Then PM, the set-wise stabilizer of M in P, is found by a
* recursive call to SetStab. PM is either all of Stab(M) in G
* or some coset Pt in G contains a permutation b which fixes M
* and Stab(M) = < PM, b >, so search Pt for a b mapping M to M by
* calling CosetStab.
*
{*****}

FUNCTION TranSetStab( G: PtrGenElt; ntupleSize: integer;
    ON: SetTypePtr; ONSize: integer;
    M: SetTypePtr; MSize: integer ): PtrGenElt;
EXTERNAL;

VAR
    P: Group;
    t: PtrGenElt;
    PM: PtrGenElt;
    PGens: PtrGenElt;
    b: PtrGenElt;
    i: integer;
    N: SetTypePtr;
    NSize: integer;

REF
    ZSize: integer;

-----

BEGIN
    P := BlkSys( G, ntupleSize, ON, ONSize );

    { CHOOSE A PERMUTATION t TO REPRESENT COSET Pt WHERE t IS
      NOT IN THE INTRANSITIVE SUBGROUP P }

    t := G;
    WHILE GroupElement( t@.Gen, P ) DO
        t := t@.NextGenElt;
    PGens := P@.Gens;
    P@.Gens := nil;
    DisposeTower( P );
    PM := SetStab( PGens, ntupleSize, ON, ONSize, M, MSize );

    { EITHER PM IS ALL OF STAB( M ) OR ELSE
      STAB( M ) = PM + PMb FOR SOME b IN COSET Pt,
      WHERE ( M )b = M }

    FindMainInverse( N, NSize, t, M, MSize, ntupleSize );
    b := CosetStab( PGens, N, NSize, ntupleSize, ON, ONSize, M,
        MSize );
    dispose( N );
    IF b <> nil THEN BEGIN
        b@.NextGenElt := PM;
        FOR i := 1 to ZSize DO BEGIN
            b@.GenInverse[ i ] := t@.Gen@[ b@.Gen@[ i ] ];
        END; { FOR }
        dispose( b@.Gen );
        b@.Gen := b@.GenInverse;
        b@.GenInverse := InvertPerm( b@.Gen, ZSize );
        PM := b;
    END; { IF }
    DispGenList( PGens );
    TranSetStab := PM;
END; { TranSetStab }

```