J. Symbolic Computation (1991) **12**, 499–516

The Computation of Normalizers in Permutation Groups

D. F. HOLT

Mathematics Institute, University of Warwick, Coventry CV4 7AL, UK

(*Received* 23 *July* 1987)

We describe the theory and implementation of an algorithm for computing the normalizer of a subgroup H of a group G, where G is defined as a finite permutation group. The method consists of a backtrack search through the elements of G, with a considerable number of tests for pruning branches of the search tree.

1. Introduction

The general problem of computing the normalizer, in a finite permutation group G, of a subgroup H has long been recognized as being unusually difficult to solve efficiently. The corresponding problem for centralizers is much easier, although this too has some bad cases. The solution to the centralizer problem is relatively simple and probably difficult to improve upon, whereas there seems to be almost unlimited scope for possible improvements to the normalizer problem. The aim of this paper is to describe some of these improvements which have been successfully implemented by the author. The ability to compute normalizers is important, not only in itself, but because it is potentially an ingredient in other algorithms, such as computing Sylow subgroups of groups or automorphisms of groups.

One attempt at a solution of the normalizer problem has been described in Butler (1983). The author's program uses the same general method, but tries much harder to keep the cpu time as low as possible. More specifically, the general idea is to impose the structure of a tree on the elements of G, and to perform a backtrack search through the tree, looking for elements of G which normalize H. At a given node in the tree, it is often possible to use group-theoretical arguments to show that none of the elements of G lying below that node can possibly normalize H, in which case we do not need to search that part of the tree; in other words, we can chop off the branch at that node, and save ourselves a lot of time. Naturally, the higher the node, the more we chop off, and the more time we save. The improvements introduced by the author have been more of these group-theoretical tests designed to prune the search tree. Of course, the tests themselves introduce certain overheads in terms of both time and space, but the experimental evidence suggests that the time saved overall is enormous in many cases, whereas space is unlikely to be a serious problem.

For example, the case in which Butler's algorithm performs worst is when H is a regular group (that is, it acts transitively, with all of its non-trivial elements acting fixed-point-freely), and G is the whole symmetric group. In fact it becomes impractical in most examples for degrees greater than about 20. The author's algorithm, on the other hand, can cope reasonably quickly with this case for degrees over 100 in many examples. As can be seen from the comparative performance statistics given at the end of this paper, it is not uniformly superior to the Butler algorithm (particularly in view of the fact that the overheads in CAYLEY may slow things down by a factor of 2 or 3), but it is much more consistent. This is largely due to its handling of regular orbits.

The algorithm described here, which is implemented in the "C" programming language, and forms part of the author's suite of programs to compute cohomological properties of finite groups, has been successfully applied in hundreds of cases, with the degree of G going up into the thousands at times. (The example of largest degree to date was in fact $G = U_7(2)$ of order $2^{21} \times 3^8 \times 5 \times 7 \times 11 \times 43 \simeq 2{\cdot}27 \times 10^{14}$ and degree 2709.) As a by-product, the same program can be used to compute centralizers, with roughly the same efficiency as other programs for this purpose. It has also been integrated into a program for computing Sylow subgroups in permutation groups, which may be a serious competitor for that used in CAYLEY, which is described in Butler & Cannon (1989). Furthermore, the author has used it on several occasions to compute automorphisms of groups, in cases which would otherwise be very difficult. These ideas will be discussed further at the end of the paper. I believe that it could be extended in the same way as in Butler (1983) to solve the problem of whether two subgroups of G are conjugate, although this has not yet been attempted.

The plan of the paper is as follows. The concepts of bases, strong generating sets and backtrack searches in permutation groups will be described briefly in section 2. This is principally to introduce notation, and the reader should preferably have some prior familiarity with these topics. Sections 3 and 4 will be devoted to descriptions of the tree-pruning tests that we use. In section 5, we shall provide more technical descriptions of the author's implementation of the material described in sections 2, 3 and 4, using the syntax of the PASCAL programming language. In section 6, we shall describe how the bases of G and H are chosen, and then, in section 7, the full algorithm will be presented concisely. The reader who is not interested in technicalities could skip sections 5, 6, and 7. Finally, in section 8, we shall discuss related algorithms, such as the computation of Sylow subgroups, and provide some performance statistics.

2. Bases, Strong Generating Sets and Backtrack Searches

The notions of bases and strong generating sets in permutation groups were first introduced in Sims (1971*a*, *b*) and are fundamental to virtually all of the existing algorithms for computing in finite permutation groups.

Let G be a permutation group acting on the set $\Omega = \{1, 2, \ldots, n\}$. For $g \in G$ and $\alpha \in \Omega$, the image of α under g will be denoted by α^g, α^G will denote the orbit of α under G, and G_α will denote the stabilizer of α in G. A sequence of points $\beta_1, \beta_2, \ldots, \beta_k$ in Ω is called a *base* for G, if its stabilizer $G_{\beta_1, \beta_2, \ldots, \beta_k}$ in G consists of the identity element only. We shall assume $\beta_1, \beta_2, \ldots, \beta_k$ is a base for G for the remainder of this section. For $0 \leq i < k$, let $G^{(i+1)} = G_{\beta_1, \beta_2, \ldots, \beta_i}$ (so that $G^{(1)} = G$). A subset S of G is called a *strong generating set* relative to the base if $\langle S \cap G^{(i)} \rangle = G^{(i)}$, for $1 \leq i \leq k$. (In other words, S contains generators for each subgroup in the stabilizer chain.) For each $1 \leq i \leq k$, let $\Delta^{(i)}$ denote the orbit of β_i under $G^{(i)}$. Then, according to the Orbit-Stabilizer Theorem, $G^{(i)}$ is in one-one correspondence with a set $U^{(i)}$ of right coset representatives of $G^{(i+1)}$ in $G^{(i)}$. The importance of the strong generating set lies in the fact that it can be used to compute the $\Delta^{(i)}$ and $U^{(i)}$ quickly. For a given $\alpha \in \Delta^{(i)}$, let $u_\alpha^{(i)}$ be the element in $U^{(i)}$ which maps

β_i to α. We shall always choose $u_{\beta_i}^{(i)}$ to be the identity element. Then every element g of G has a unique representation of the form

$$g = u_{\alpha_k}^{(k)} \dots u_{\alpha_2}^{(2)} u_{\alpha_1}^{(1)},$$

for suitable $\alpha_i \in \Delta^{(i)}$, and so we have an easily computable one–one correspondence between the elements of G and the set T of k-tuples $(\alpha_k, \dots, \alpha_1)$.

If S is a strong generating set for G, we will denote $S \cap G^{(i)}$ by $S^{(i)}$. For each element g of S, we store an associated integer i, which is the largest i for which $g \in S^{(i)}$, and so $S^{(i)}$ can easily be calculated.

Let us now order each $\Delta^{(i)}$ arbitrarily, except that β_i must come first. This induces a reverse-lexicographical ordering on T, and hence an ordering on the elements of G, in which the identity element comes first and, for each i, the elements of $G^{(i)}$ precede those in $G \backslash G^{(i)}$. [This is not quite the same ordering as that used in Butler (1983), which is an ordering by base-image, but this makes little difference to the algorithm.] This is the order in which we shall be searching through the elements of G in our quest for elements which normalize the subgroup H.

The backtrack search procedure which we shall now summarize is common to many algorithms in group theory and combinatorics. It has been described, for example, in Butler (1983), Leon (1984) and Cannon (1985). We first define a tree whose nodes are arranged in $k+1$ layers. Layer 0 (which is never actually used) has a single node labelled (), and, for $1 \leq i \leq k$, layer i has nodes labelled $(\alpha_i, \dots, \alpha_1)$, where $\alpha_j \in \Delta^{(j)}$ for $1 \leqslant j \leq i$. A node x is joined to a node y in the next layer down if y is derived from x by adjoining an element at the front of the sequence. The nodes in the bottom layer k correspond to elements of G. Nodes at higher levels i correspond to indeterminate elements of G, for which only the images of the first i base points are known. The search starts at the top of the tree, and proceeds down. At each node, at level i, we apply as many tests as possible in an attempt to prove that any element of G with the given images of the first i base points cannot possibly lie in the normalizer of H (or that, if it does, then we know about it already). These tests will be described in the following two sections. If we succeed in this aim, then we need go down no further from this node, and we can proceed to the next node at level i. If we arrive at layer k, then we have a specific element of G, and we can test directly whether this normalizes H. The required normalizer $N_G(H)$ will be built up as we proceed. We start by putting $N = H$, and every time we find a new element $g \in N_G(H)$, we replace N by $\langle N, g \rangle$. Of course, the tests at the higher levels of the tree should rule out elements which are already in N, as well as those which cannot possibly lie there.

3. Tests Based on Orbit Structure

The complete algorithm is divided into two phases. During the first phase, the bases of G and H are chosen, and various information that will be used in the tests during the main search is computed and stored. The second phase consists of the search itself. Of course, in all but the smallest of examples, the vast majority of the time is taken up with the search, and the first phase is simply an overhead resulting from a complex algorithm. The criteria underlying the choice of bases will become apparent from the descriptions of the individual tests in this and the following section.

In $g \in N_G(H)$, then g must permute the orbits of H. More generally, if g maps the sequence of points $\beta_1, \beta_2, \dots, \beta_i$ to $\alpha_1, \alpha_2, \dots, \alpha_i$, then it must map the orbits of $H_{\beta_1, \beta_2, \dots, \beta_j}$ to those of $H_{\alpha_1, \alpha_2, \dots, \alpha_j}$ for $0 \leq j \leq i$. When we are at level i in the search tree, we are in a very convenient position to test that this condition applies, when $\beta_1, \beta_2, \ldots, \beta_i$ is the sequence of the first i base points. This will be one of our tests to apply at a node in the search tree. (It is, in fact, Test 3 in the list of tests given in section 7 below.) To facilitate this test, we compute and store all of the orbits of each of the subgroups $H^{(i+1)}$ in the stabilizer chain of H before we start the main search. At a given point during the search, when we are considering a potential $g \in N_G(H)$ that maps $\beta_1, \beta_2, \ldots, \beta_i$ to $\alpha_1, \alpha_2, \ldots, \alpha_i$, we need to compute the corresponding orbits of $H_{\alpha_1, \alpha_2, \ldots, \alpha_i}$, which involves changing the base of H to $\alpha_1, \alpha_2, \ldots, \alpha_i, \ldots$ Further technical details will be provided in section 5, but it is not difficult to see how to apply the test once the relevant orbits have been computed. In fact, there are three ways in which this test can be failed: (i) a point in an orbit maps onto a point in an orbit of different length; (ii) two points in the same orbit map onto points in different orbits; (iii) two points in different orbits map onto points in the same orbit.

It is also, in principal, possible to apply this test with $\beta_1, \beta_2, \ldots, \beta_i$ replaced by any permutation of itself. This is technically more difficult, since it is not feasible to store in advance all orbits of all of the possible stabilizers of subsets of $\{\beta_1, \beta_2, \ldots, \beta_i\}$. We content ourselves with the following compromise, which seems fairly effective in a number of examples. If β_i lies in a basic orbit $\Delta_H^{(j)}$ for some $j \leq i$, then we compute the orbits of the stabilizer of the points $\beta_1, \beta_2, \ldots, \beta_{j-1}, \beta_i$ *during the search*, by conjugating the orbits of $H^{(j+1)}$ by an element of $H^{(j)}$ that maps β_j to β_i, and we do the same for the potential images $\alpha_1, \alpha_2, \ldots, \alpha_{j-1}, \alpha_i$. We can then carry out the corresponding orbit permutation test. Of course, these extra orbit computations are themselves somewhat time consuming but, where they do not help they result in only a small increase in cpu time, whereas in favourable cases they can result in a dramatic improvement.

Let us now illustrate the use of these tests with a small example. Let H (which is the direct product of two copies of the dihedral group of order 12) be generated by the permutations (1,2,3,4,5,6), (7,10,9,8,11,12), (2,6)(3,5) and (10,12)(9,11), and suppose that $\beta_1 = 1, \beta_2 = 2$ and we are considering an element $g \in G$ with $1^g = 7$. Then g will have to map the orbit $\{1,2,3,4,5,6\}$ of H to $\{7,8,9,10,11,12\}$. We now change the base of H such that $\beta_1 = 7$, and compute the orbits of H_7. We may now try $2^g = 8$, but the orbit $\{2,6\}$ of H_1 has length 2, whereas the orbit $\{8\}$ of H_7 has length 1, so this is impossible. Trying $2^g = 9$, we find that the orbit $\{2,6\}$ of H_1 is mapped to the orbit $\{9,11\}$ of H_6, and so we must have $6^g = 11$. (This means, incidentally, that 6 is the correct choice for β_3.) By conjugating H_1 by $(1,2,3,4,5,6) \in H$, we find that H_2 has an orbit $\{1,3\}$ which must be mapped to the orbit $\{7,11\}$ of H_9, and so we have $3^g = 11$, which is impossible since $6^g = 11$. Thus $2^g = 9$ is impossible. Moving on to $2^g = 10$, $\{2,6\}$ is mapped to $\{10,12\}$ and so $6^g = 12$. Conjugating by (1,2,3,4,5,6) as before, we find that $\{1,3\}$ is mapped to the orbit $\{7,9\}$ of H_{10}, and so $3^g = 9$. We then quickly deduce in a similar way that $5^g = 11$ and $4^g = 8$. Now, since $H \subseteq N_G(H)$ and $H_{7,8,9,10,11,12}$ has an orbit $\{1,2,3,4,5,6\}$, we can effectively assume that $7^g = 1$. Trying $10^g = 2$, we get $9^g = 3$, $8^g = 4$, $11^g = 5$ and $12^g = 6$ using orbital arguments as before, and we have completely constructed g, which does indeed turn out to normalize H. Apart from $10^g = 6$, which is essentially equivalent, since $\{6,10\}$ is an orbit of $H_{1,7,8,9,10,11,12}$, all other possibilities for 10^g turn out to be impossible in the same way that $2^g = 8$ or 9 was impossible above.

The reader may justifiably object that the above example can be explained much more clearly by observing that H is acting as a group of automorphisms of a graph consisting of two disjoint hexagons. Indeed, the algorithm could be implemented by bringing in graph automorphisms in this fashion, and it would even be somewhat more efficient on certain examples, but we have preferred to avoid this, since it involves so much additional machinery.

Concerning the initial choice of bases for G and H, the main lesson to be learnt from the discussion above is that when a base point β_i of H (and G) has been chosen, then as many as possible immediately subsequent base points of G should be chosen from $\Delta_H^{(i)}$. However, when we have to choose a base point γ_i for H from one of a number of distinct orbits of $H^{(i)}$, then it can be difficult to know how to select the orbit from which to choose it. This can have a crucial effect on the time taken by the search, but unfortunately it does not seem easy to find a general rule that works well on all examples. The author's current strategy is to choose the shortest orbit on which $H^{(i)}$ acts faithfully, if any, but otherwise to choose the longest orbit. However, the user has the option of overriding this choice interactively.

4. Tests Based on Induced Automorphisms

Two tests will be described in this section. They both make use of the fact that an element of G that normalizes H must induce an automorphism of H, by conjugation. The idea is to use the first few base images to deduce enough properties of this automorphism to enable the machine to compute some of the later base images. When this is possible, it clearly represents a considerable reduction in the total search time.

The first of these tests is based on regular actions of certain sections of H. If H acts regularly on the points, then the tests described in section 3 yield no information whatsoever. However, in this case, for any point α, the group $N_G(H)_\alpha$ is isomorphic to a subgroup of Aut(H) (and in case G is the symmetric group, we get the whole of Aut(H)), and so the normalizer problem is really equivalent to the problem of computing an automorphism group. The only properties of automorphisms that we attempt to use are that an automorphism maps a subgroup onto a subgroup of the same order and an automorphism is uniquely determined by its action on the generators, but these will be enough to render the algorithm practical for groups of degree a few hundred (depending on G), in the case of a regular subgroup H. More generally, we can apply this test to any regular orbit of $\Delta_H^{(i)}$ for any i.

As an example, suppose that H is the elementary abelian group of order 9 with generators $h=(1,4,7)(2,5,8)(3,6,9)$ and $k=(1,2,3)(4,5,6)(7,8,9)$. Suppose also that $\beta_1=1$, $\beta_2=4$ and we are considering $g\in G$ with $1^g=1$ and $4^g=5$. Then, since $1^{g^{-1}hg}=5$, if $h^g\in H$ then $h^g=hk$. It then follows that $5^{hk}=5^{g^{-1}hg}=7^g=9$, and so 7 is the correct choice for β_3. The choice of β_4 is unimportant, so suppose that $\beta_4=2$ and we attempt $2^g=2$. Then $k^g=k$ and we now know the action of g on the whole of H and we quickly deduce $3^g=3$, $5^g=6$, $6^g=4$, $8^g=7$ and $9^g=8$. The case in which H is elementary abelian is admittedly the easiest, since every sensible choice of base images will extend to an element of N(H). This is not always the case. If, for example, H is abelian of order 8 with generators $h=(1,2)(3,4)(5,6)(7,8)$ and $k=(1,5,3,7)(2,6,4,8)$, then we may initially try $\beta_1=1$, $\beta_2=2$, $1^g=1$ and $2^g=3$, which gives $h^g=k^2$. It is not immediately clear (to the computer) that this does not extend to an automorphism of H, but this will eventually be deduced from the fact that no choice of 3^g will work, and so the time lost will not be enormous.

We turn now to the second of the tests to be described in this section. The idea is as follows. Suppose that H has orbits $\Delta_1,\ldots,\Delta_r$ and it acts faithfully on Δ_1. Suppose further that the action of the candidate for $g\in N(H)$ on Δ_1 has been determined. Then the automorphism of H induced by g is also determined. It follows that, for each Δ_i with $i>1$, as soon as the action of g on one point of Δ_i has been specified, then the action can be deduced on the whole of Δ_i. If the action on Δ_1 is not faithful, then the same is true provided that the kernel of the action on Δ_1 is a subgroup of the kernel of the action on Δ_i.

As an example, suppose that H is isomorphic to the dihedral group of order 14 with generators

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

and

$$k = (2,7)(3,6)(4,5)(8,10)(11,14)(12,13),$$

and we are trying $\beta_1 = 1$, $\beta_2 = 2$, $1^g = 1$ and $2^g = 3$. Then, using orbit tests as in section 3, we might choose $\beta_3 = 7$, $\beta_4 = 3$, $\beta_5 = 6$, $\beta_6 = 4$, $\beta_7 = 5$ and deduce $7^g = 6$, $3^g = 5$, $6^g = 4$, $4^g = 7$, $5^g = 2$. We now know the action of g on the faithful orbit $\{1,2,3,4,5,6,7\}$ of H, and so we can compute the automorphism of H induced by conjugation by g, which is $h^g = h^2, k^g = k$. The best choice for β_8 is, in fact, 9, since $\text{Fix}(H_1) = \{1,9\}$, and we can immediately deduce that $9^g = 9$. We then get $10^g = 9^{hg} = 9^{g^{-1}hg} = 9^{h^2} = 11$, and we deduce the images of each of the other points in this fashion. In fact, the author's current implementation is not quite so clever, so it would probably choose $\beta_8 = 8$. It would then try all possibilities for 8^g. From $8^g = 8$, it would deduce $9^g = 10$, and reject this, since it cannot permute the orbits of H_1. Eventually, it would try $8^g = 14$ and deduce $9^g = 9$, which is correct.

5. Technicalities and Implementation

In this section we shall discuss the technicalities involved in the implementation of the material described in sections 2, 3 and 4. We start with a formal description of the backtrack-search algorithm described in section 2. This will be amplified in section 7. We shall use the syntax of PASCAL, but with some additions to the operations available on sets. We shall use the normal mathematical symbols "$\cup$" and "$\cap$" rather than "+" and "$*$", and we shall use $|S|$ to denote the cardinality of the set S. In order to allow us to loop over the elements of a set, we introduce two functions *first* and *next*. If S is a set, then *first*(S) will denote some arbitrary first element of S, and for $x \in S$, *next*(x) will denote the next element, with *next*(x) = 0 on the last element x of the set.

We shall now change some of the notation to a more "computerish" style. *n_pts* will denote the number of points being permuted, and *len_G_base* the number of base points. The base points will be *G_base*[1], . . ., *G_base*[*len_G_base*]. *G_base_no*[1:*n_pts*] will be the inverse function to *G_base* on the base, and equal to zero on points not in the base. (Later on in the paper we shall use similar names for the functions of the subgroup H, where "G" is replaced by "H" in the names.) *testperm*[1:*n_pts*] will be the permutation that we are currently testing for membership of $N_G(H)$. The variable *level* will denote the level of the current node in the search tree. We assume that we have a function *image* defined for computing the value of *testperm* at *G_base*[*level*]. This is, of course, merely the image of *G_base*[*level*] under the permutation $u^{(i)}_{\alpha_i} \ldots u^{(2)}_{\alpha_2} u^{(1)}_{\alpha_1}$, where $i = level$. The points $\alpha_1, \ldots, \alpha_i$ in this expression will be denoted by *base_im*[1], . . ., *base_im*[*level*], and *base_im*[*j*] will be zero for values of j larger than the current level. In order to move on to the next value of α_i at *level* = i, we need functions *next*(i,*point*). We assume, without going into details, that these functions are defined when $\Delta^{(i)}$ is computed. It is convenient to put *next*(i,0) = *G_base*[i], and *next*(i,α_i) = 0 for the last point α_i in the orbit, so as to achieve a circular linked list. We assume that subgroups K of G (like H and N) are defined by sets of strong generators Sg(K). In practice, this will mean storing some additional information for K, such as Schreier vectors, but we omit details of this, since it is reasonably well known. We can then test arbitrary permutations for membership of K [using the algorithm "Strip" described in Cannon (1985)] and so we can define a Boolean function *member(perm,*K). Finally, the variable *firstmoved* will denote the least value of i for which α_i is not equal to β_i, or, in other words, *base_im*[*i*] is not equal to 0 or *G_base*[*i*]. The order of the search ensures that, if *firstmoved* $= i$, then $N^{(j)} = N_G(H)^{(j)}$, for all $j > i$. Thus, whenever we find a new element of N, we can immediately return to *level = firstmoved*. A "*" will be used for permutation multiplication.

```
for level:=1 to len_G_base do base_im[level]:=0;
level:=1; Sg(N):=Sg(H); firstmoved:=len_G_base;
while (level>0) do
begin base_im[level]:=next(level,base_im[level]);
  if base_im[level] =0 then
  begin level:=level−1;
    if level<firstmoved then firstmoved:=level;
  end else
  begin basept:=G_base[level]; testperm[basept]:=image(basept);
    if all tests passed at this node then
    begin if level<len_G_base then level:=level+1 else
      begin for i:=1 to n_pts do if G_base_no[i]=0 then
          testperm[i]:=image(i);
        ok:=true; h=first(Sg(H));
        while (ok and h≠0) do
        begin if not member(testperm⁻¹*h*testperm,H)
              then ok:=false;
          h:=next(h);
        end;
        if ok then
        begin Sg(N):=Sg(N)∪[testperm];
          while level>firstmoved do
          begin base_im[level]:=0; level:=level−1
          end;
        end;
      end;
    end;
  end;
end;
```

We turn now to the implementation of the orbital tests discussed in section 3. The variables *H_base*, *H_base_no* and *len_H_base* correspond to the same variables with "G" in place of "H", and we assume that *H_base* is a subsequence of *G_base*. It is convenient to define *H_base_no*[β] to be negative when β is in the base of G but not that of H, so that we can locate how far we are along the H-base at a given level. For example, if the base of G is 1,3,4,6,8 and that of H is 1,4,6 then the values of *H_base_no* on 1,3,4,6,8 will be, respectively, $1, -1, 2, 3, -3$. The variable *HeqG_no* will denote the highest value of i, for which the basic orbit $\Delta_H^{(j)}$ of β_j under H is the same as $\Delta^{(j)}$,for $1 \le j \le i$. For example, let G be the full symmetric group on five symbols. Then, in the four cases in which H is, respectively, Alt(5), a Frobenius group of order 20, a dihedral group of order 10, and a cyclic group of order 3, *HeqG_no* is equal to 3, 2, 1 and 0. Since the main search starts off with N = H, it is only really necessary to conduct the search on the nodes of the tree at levels larger than *HeqG_no*. Furthermore, we will make no use of the orbits of $H^{(j)}$ for $j \leq$ *HeqG_no*, since they are likely to be the same as those of $G^{(j)}$, and they are unlikely to yield useful information.

The required orbital information will be computed by a procedure *allorbs*(S,*orb_no*,*orb_len*,*orb_rep*), where S is a set of permutations, and *orb_no*, *orb_len* and *orb_rep* are three arrays whose values are computed by the procedure. The orbits under the group generated by S are computed. *orb_no*[0: *n_pts*] assigns an orbit number in the range $1, \ldots, m$ to each point, where *orb_no*[0] is used to denote the total number m of orbits. *orb_len*[1: m] gives the length of each orbit, and *orb_rep*[1: m] gives a representative point from each orbit. Since the computation of orbits in permutation groups is well understood, we need not discuss the code for *allorbs*.

From now on, we shall denote the base of H by $\gamma_1, \gamma_2, \ldots$ and its image under *testperm* by $\delta_1, \delta_2, \ldots$. We shall need the orbital information for the subgroups $H^{(i+1)} = H_{\gamma_1, \gamma_2, \ldots, \gamma_i}$, where $i+1$ ranges from *HeqG_no*+1 to *len_H_base*. We therefore need two-dimensional arrays *H_orb_no*, *H_orb_len* and *H_orb_rep*. (These are best thought of as arrays of arrays; for example, we have

$$H\_orb\_no[HeqG\_no+1 : len\_H\_base][0 : n\_pts].)$$

They can be computed before we start the search. When we are at level i in the search tree, we shall also need the corresponding orbital information for the subgroups $\mathrm{Im}H^{(j+1)} = H_{\delta_1, \delta_2, \ldots, \delta_j}$, for the appropriate values of j. (To be precise, j will range from *HeqG_no* to the larger of abs(*H_base_no*[*G_base*[i]]) and *len_H_base* − 1.) This will have to be computed during the course of the search, and to do this we need to change the base of H to $\delta_1, \delta_2, \ldots, \delta_j$. [An algorithm for changing the base in a permutation group is described by Sims (1971*a*).] This information will be stored in the arrays *imH_orb_no*, *imH_orb_len* and *imH_orb_rep*. The actual orbital maps induced by *testperm* will be stored in arrays *H_orb_im*[i][1: m], which are defined on the m orbits of $H^{(i)}$, where m = *H_orb_no*[i][0]. Since these maps will be updated continually, it is vital to know the value of *level* at which each entry was made. This is stored in arrays *deftime*[i][1: m]. To be precise, whenever we make an entry *H_orb_im*[i][j] = k, we put *deftime*[i][j] equal to the current value of *level*. Then, whenever *base_im*[*level*] is changed, we must delete all entries in *H_orb_im* for which the corresponding value of *deftime* is greater than or equal to *level*.

As described in section 3, there are three possible causes of failure in this test. Provided that all of the information described above is available, the code to test these conditions is straightforward to write. It must be stressed that, at level i, the tests are made on the orbits of $H^{(j)}$ for as many values of j as possible at this level.

There is one further test, of a slightly different nature, that can be applied with this information. Suppose there is an element of $N_G(H)$ that maps the H-base point γ_j to some point δ_j. Then, since $H \subseteq N_G(H)$, there is an element of $N_G(H)$ mapping γ_j to ε_j, for any ε_j in the orbit of $\mathrm{Im}H^{(j)}$ with the number *imH_orb_no*[j][δ_j]. We may therefore reject δ_j as a possible image of γ_j in the search, unless

$$\delta_j = imH\_orb\_rep[j][imH\_orb\_no[j][\delta_j]].$$

In fact, we only apply the test in this form when *level* > *firstmoved*. When *level* = *firstmoved*, we apply a slightly stronger test, by computing all of the orbits of $N^{(i)}$, for i = *level*, and insisting that β_i is mapped onto the chosen representative of the orbit onto which it is mapped. (In other words, we are applying the test to N rather than to H.) To apply it to N at higher levels would involve making base changes in N as well as in H, which would considerably complicate the program. These tests (which comprise Test 2 in the list in section 7), correspond to one of the "first element in coset" tests described on pp. 328–330 of Leon (1984).

It is sometimes worthwhile to perform further orbital tests, based on permutations of the first few base points. Suppose that $G\_base[level] \in \Delta_H^{(i)}$ for some i. Then, for each k satisfying $i<k\leq level$, we compute a permutation $h \in H^{(i)}$ that maps $G\_base[i]$ to $G\_base[k]$, and conjugate the orbits of $H^{(i+1)}$ by h, to give the orbits of the stabilizer of $G\_base[k]$ in $H^{(i)}$. This is done during the course of the search, since storing the information in advance would take up a great deal of space. We also compute the corresponding orbits of the stabilizer of $testperm[G\_base[k]]$ in the current group $H^{(i)}$. We then check that the orbital mapping induced by *testperm* on the points $G\_base[l]$ for $i\leq l\leq level$ is consistent, in the usual way. The problem with this procedure is that it is somewhat time consuming to carry out and it often yields no new information. In some cases, however, it results in a dramatic improvement, and so one has to make a careful decision about when it should be attempted. The author's current implementation has options to employ several different possible strategies in this respect, since the optimal approach unfortunately seems to vary from example to example.

We turn now to a more technical discussion of the automorphism tests described in section 4. As before, we assume that *testperm* maps the H-base points $\gamma_1, \gamma_2, \ldots, \gamma_i$ to $\delta_1, \delta_2, \ldots, \delta_i$. For $1\leq i\leq len\_H\_base$, let $F^{(i)} = \mathrm{Fix}(H^{(i+1)})\cap\Delta_H^{(i)}$, where "Fix" denotes the fixed point set of a subgroup, and let $\mathrm{Im}F^{(i)}$ be defined similarly in terms of $\mathrm{Im}H^{(i+1)}$. Note that $\gamma_i \in F^{(i)}$ and $\delta_i \in \mathrm{Im}F^{(i)}$. If $g\in H^{(i)}$ satisfies $\gamma_i^g = \gamma \in F^{(i)}$, then g permutes the set $F^{(i)}$. It follows that, if $FH^{(i)}$ is the subgroup of $H^{(i)}$ that permutes $F^{(i)}$, then $FH^{(i)}$ acts regularly on $F^{(i)}$, and the same thing applies for the corresponding subgroup $\mathrm{Im}FH^{(i)}$ of $\mathrm{Im}H^{(i)}$ that permutes the set $\mathrm{Im}F^{(i)}$. The permutation *testperm* maps $F^{(i)}$ onto $\mathrm{Im}F^{(i)}$, and conjugates $FH^{(i)}$ to $\mathrm{Im}FH^{(i)}$, and so we can apply tests based on induced automorphisms.

The choice of the bases of H and G will be discussed in the next section, but, in this situation, we always choose as many G-base points as possible following γ_i to lie in $F^{(i)}$. (These will not be H-base points.) Let us call them $\rho_1, \rho_2, \ldots, \rho_k$, and let g_j be an element in $FH^{(i)}$ satisfying $\gamma_i^{g_j} = \rho_j$, for $1\leq j\leq k$. Whenever we have chosen $\rho_1, \ldots, \rho_j$, for some j, we compute the orbit $F_j(\subseteq F^{(i)})$ of γ_i under $\langle g_1, \ldots, g_j\rangle$ and remember its size. If possible, we next choose $\rho_{j+1}\in F_j$. We can then choose $g_{j+1}\in\langle g_1, \ldots, g_j\rangle$, and in this case we store the word in $g_1, \ldots, g_j$ which represents g_{j+1}. This information is used in the following manner in the course of the search. When we are at the node corresponding to γ_i, we can compute the set $\mathrm{Im}F^{(i)}$, and test that it has the same size as $F^{(i)}$. This is part of Test 6 in section 7. When we are at a node corresponding to a ρ_j for which we have no stored word, we compute the set $\mathrm{Im}F_j$ (defined in the obvious manner), and test that it has the same size as F_j. This is Test 4 in section 7. When we are at a node corresponding to a ρ_{j+1} for which we have a stored word, we compute the corresponding word using elements h_j in place of g_j, where h_j is chosen such that $\delta_i^{h_j}$ is the image of ρ_j under *testperm* (h_j can be computed using the current Schreier vector of H). The image of ρ_{j+1} under *testperm* can then be computed as the image of δ_i under this word. This is Test 1a in section 7.

The data that we need to remember to carry out the above test is stored during the initial choice of base. One possibility is to have an array *wordinfo* to hold the strings expressing g_{j+1} in terms of $g_1, \ldots, g_j$, and possibly the lengths of the F_j. Another array, *pointtype*[1: *len_G_base*], can be used to indicate, in some convenient fashion, to which category each G-base point belongs, and to point, if necessary, to the relevant string in *wordinfo*. In order to be specific, let us use the value of *pointtype* modulo 16 to denote the category, and the integral part of *pointtype*/16 for the pointer. So far, we have four categories 0, 1, 3 and 2 corresponding, respectively, to nothing in particular, a point γ_i for which we will compute a set $\mathrm{ImF}^{(i)}$, a point ρ_j for which we will compute a set ImF_j, and a point ρ_{j+1} for which we can compute the image under *testperm.*

Finally, we consider the second of the automorphism tests described in section 4. Once again, we need to apply this idea also to the stabilizers $\mathrm{H}^{(i)}$, and we shall take Δ_1 to be the basic orbit $\Delta_{\mathrm{H}}^{(i)}$. The policy will always be to choose as many G-base points $\gamma_i, \rho_1, \rho_2, \ldots, \rho_j$ as possible that lie in Δ_1. (Here $j \geq 0$, and the ρ_j might or might not be H-base points.) This ensures that the action of any $g \in \mathrm{G}$ is determined on the whole of Δ_1 by its action on these base points. If $i > j \geq 0$ and $\Delta_{\mathrm{H}}^{(i)} \subset \Delta_{\mathrm{H}}^{(j)}$, then it is convenient to restrict ourselves to orbits Δ_i of $\mathrm{H}^{(i)}$ that lie within $\Delta_{\mathrm{H}}^{(j)}$ (since the points outside of $\Delta_{\mathrm{H}}^{(j)}$ will come within the scope of the search for orbits of $\mathrm{H}^{(j)}$). At this point, we search for a suitable Δ_2 satisfying the required condition on the kernel of the action; namely, the stabilizer in $\mathrm{H}^{(i+1)}$ of $\rho_1, \rho_2, \ldots, \rho_j$ must fix Δ_2 pointwise. We then choose as many G-base points $\sigma_1, \sigma_2, \ldots, \sigma_k$ as possible in Δ_2. (These will definitely not be H-base points.) For each such σ_t with $t > 1$, we compute an element of $\mathrm{H}^{(i)}$ mapping σ_1 to σ_t, and remember this element as a string in the (original) generators of H. We can then go on to seek further orbits Δ_i satisfying the required conditions. (Of course, in many examples there will be no suitable Δ_i or, if there are some, then there may be no G-base points to be found within them.)

The information described in the preceding paragraph is computed and stored during the initial choice of base. The arrays *pointtype* and *wordinfo* can again be used to record this data. We now have some more categories of point. Points of type ρ_j, which are the last G-base point in a basic H-orbit will lie in category 4, 6 or 7, depending on whether they already lie in category 0, 2 or 3. Points of type σ_1 lie in category 8, and those of type σ_j for $j > 1$ in category 10. Note that the G-base points in categories with numbers equal to 2 modulo 4 are exactly those for which the image under *testperm* can be computed. For points in category 4, 6, 7 or 8, it is also important to be able to locate the H-base point γ_i at the beginning of the corresponding basic H-orbit; to this end, we shall introduce an additional array *backpointer*, and put *backpointer*[*level*] equal to *i*. Similarly, for points σ_j in category 10, we have to be able to locate *G_base_no*[σ_1], and *backpointer* can be used for this purpose.

We now describe how this data is used during the search. Since the base of H is changed frequently during the course of the search, and we also need the original generators of H from time to time, it is necessary to keep a copy of the original generators. Let Old_Sg(H) denote this original strong generating set, where Sg(H) will always denote the current set. Suppose first that we are at a node corresponding to a point in category 4, 6 or 7. Then we compute the image of our current partial expression $u_{\alpha_k}^{(k)} \ldots u_{\alpha_2}^{(2)} u_{\alpha_1}^{(1)}$ for *testperm* on the whole of Δ_1, and check that this image is precisely the image orbit $\mathrm{Im}\Delta_1$ of δ_i under the current $\mathrm{H}^{(i)}$. For each $g_j \in \mathrm{Old\_Sg(H)}^{(i)}$, we now conjugate g_j by *testperm*, and compute the image $\mathrm{Im}g_j$ on $\mathrm{Im}\Delta_1$, whilst testing this restriction of $\mathrm{Im}g_j$ for membership of the restriction of $\mathrm{H}^{(i)}$ to $\mathrm{Im}\Delta_1$. This is a very useful partial test to confirm that *testperm* actually lies in the normalizer of H. If this test does not fail, then we get a word for each $\mathrm{Im}g_j$ in the (current) generators of H, which is accurate on $\mathrm{Im}\Delta_1$, but it will also be accurate on each $\mathrm{Im}\Delta_i$ for which the appropriate condition on the kernel holds. In practice, we multiply this word out in full at this stage, and store the resulting permutation (which we shall continue to refer to as $\mathrm{Im}g_i$). The above test is Test 7 in section 7.

Secondly, suppose that we are at a node in the search tree corresponding to a G-base point in category 8, and let τ_1 be the proposed image of σ_1 under *testperm.* Then we check that the required kernel condition holds for the orbit $\mathrm{Im}\Delta_i$ of τ_1 under the current $H^{(i)}$, which merely involves checking that the generators of the relevant stabilizer in $H^{(i)}$ fix $\mathrm{Im}\Delta_i$ pointwise. This is Test 5 in section 7. Finally, suppose that we are at a node corresponding to a base point σ_j in category 10. Then we extract the word in Old_Sg(H)$^{(i)}$ that we stored for a permutation taking σ_1 to σ_j, and compute the same word with the Img_i in place of g_i. The image of τ_1 under this word will be the image of σ_j under *testperm.* This is Test 1b in section 7.

6. The Choice of the Bases

In this section we shall summarize the algorithm for the choices of the bases for G and for H and the computation of associated data, in the light of the demands imposed upon us by the tests described in the preceding sections. The base points will be chosen in order for both G and H, and all H-base points will also be G-base points. In practice, G and H will have initial bases, and so these choices will involve base changes. We assume that we have a function *base_change*(K,x,i) available, which changes the ith base point of the group K to x without changing the first $i-1$ base points, and makes the appropriate changes to Sg(K). The variable i represents the number of H-base points found so far and *lev* the number of G-base points found so far. *depth* represents the current degree of embedding of the basic H-orbits, and *baseno*[*depth*] is the number of first H-base point at that depth. The function *fix* is computable as the set of points in the orbits of the subgroup of length 1. The operations below that involve intersections with the $\Delta_H^{(j)}$ are straightforward to carry out, since the relevant orbital information for H will have been stored.

```
i:=lev:=0; depth:=0;
Zero the arrays G_base_no, H_base_no, pointtype, backpointer;
allorbs(Sg(H)$^{(1)}$,H_orb_no[1],H_orb_len[1],H_orb_rep[1]);
S:=$\Omega$; i1:=i+1;
while Sg(H)$^{(i1)}$ $\neq$ [ ] do
begin i:=i1; lev:=lev+1; i1:=i+1;
    Choose an orbit of H$^{(i)}$ of length>1 that lies within S, and choose any point x
    in this orbit;
    base_change(H,x,i); H_base_no[x]:=i;
    base_change(G,x,lev); G_base_no[x]:=lev;
    allorbs(Sg(H)$^{(i1)}$,H_orb_no[i1],H_orb_len[i1],H_orb_rep[i1]);
    depth:=depth+1; baseno[depth]:=i;
    S:=$\Delta_H^{(i)}$;F$^{(i)}$:=S$\cap$fix(H$^{(i1)}$);
    oldlev:=lev;
    Choose as many G-base points as possible in the set F$^{(i)}$, and increase lev
    accordingly. Store the relevant data in the arrays pointtype and wordinfo, as
    described in Section 4;
    If any such points are found, then pointtype[oldlev]:=1;
    The Boolean variable nomoreorbs is now set true if and only if H$^{(i1)}$ has no
    orbits of length>1 that lie within S;
    j:=i;
    while nomoreorbs and depth>0 do
```

```
  begin if pointtype[lev] mod 8<4 then
    pointtype[lev]:=pointtype[lev]+4;
    backpointer[lev]:=j;
    depth:=depth−1;
    if depth>0 then begin k:=baseno[depth]; S:=$\Delta_H^{(k)}$ end
          else S:=$\Omega$;
    Look for orbits $\Delta$ of $H^{(i)}$ within S which are fixed pointwise by each element
    of $Sg(H)^{(i1)}$; choose as many G-base points as possible within such $\Delta$ and
    store the relevant data in the arrays pointtype, backpointer and wordinfo, as
    described in Section 4;
    Set variable nomoreorbs as before;
    j:=k;
  end;
end;
```

For simplicity, we have omitted the parts of the above code that set the value of *HeqG_no*. This presents no difficulty, since we merely have to look out for the first G-base point for which the basic orbit differs from the basic H-orbit. One can also adapt the code to check that H is indeed a subgroup of G, and to recognize exceptional cases like $H = 1$ and $H = G$.

Finally, observe that the same G-base point may be put into category 4, 6 or 7 more than once in the course of the algorithm but, each time, it will point back to an earlier H-base point.

6. The Main Searching Algorithm

This is an elaboration of the basic searching algorithm, as presented in section 2. We start with explanations of a few points. As mentioned in section 3, when *level=firstmoved*, we make use of orbits of N. The idea of putting the representative of the orbit containing the base point to be 0, is to avoid finding elements that already lie in N. The procedure *calc_N_ orbs*(*x*) consists of the following three lines of code.

```
S:=$Sg(N)^{(x)}$;
allorbs(S,N_orb_no,N_orb_len,N_orb_rep);
N_orb_rep[N_orb_no[G_base[x]]]:=0;
```

For simplicity, we use the same base for N as for G, although this is likely to be unnecessarily large in practice. (For example, if *pointtype*[*level*] $\equiv 2 \pmod 4$, then *G_base*[*level*] is redundant as an N-base point.)

The variable *orbim_no* is used to record how far along the H-base we are. If we are between two H-base points, it takes the value of the following one. It is chiefly important because the orbit tests are carried out on the orbits of $H^{(j)}$, where j ranges from *HeqG_no*$+1$ to *orbim_no*. The procedure *update_orbims*(*level*) is used, as described in section 5, to zero those values of *defim*[*i*][*j*] for which *deftime*[*i*][*j*] $\geq$ *level*, for all appropriate *i*. The function *calc_image*(*level*) is used to compute the value of *testperm* on the current G-base point, as described in section 5, whenever this is possible; that is, whenever *pointtype*[*level*] $\equiv 2 \pmod 4$. We shall omit the details of this process.

```
Zero all values of defim and deftime;
Old_Sg(H):=Sg(H); $Img_i:=g_i$ for each $g_i \in$ Old_Sg(H);
for level:=1 to HeqG_no do
  testperm[G_base[level]]:=base_im[level]:=G_base[level];
for level:=HeqG_no+1 to len_G_base do base_im[level]:=0;
level:=HeqG_no+1; Sg(N):=Sg(H);
firstmoved:=len_G_base; calc_N_orbs(len_G_base);
while (level>HeqG_no) do
begin basept:=G_base[level];
  if H_base_no[basept]>0 then orbim_no:=H_base_no[basept]
  else orbim_no:= -H_base_no[basept]+1;
  if orbim_no>len_H_base then orbim_no:=len_H_base;
  update_orbims(level);
  base_im[level]:=next(level,base_im[level]);
  if base_im[level]=0 then
  begin level:=level-1;
    while pointtype[level] mod 4=2 do
    begin base_im[level]:=0; level:=level-1 end;
    if level<firstmoved then
    begin firstmoved:=level; calc_N_orbs(level) end;
  end else
  begin im_basept:=image(basept);
    if all tests passed at this node then
    begin testperm[basept]:=im_basept;
      if level<len_G_base then
      begin level:=level+1;
        if pointtype[level] mod 4=2 then
        testperm[G_base[level]]:=calc_image(level);
      end else
      begin for i:=1 to n_pts do if G_base_no[i]=0 then
          testperm[i]:=image(i);
        ok:=true; h:=first(Sg(H));
        while ok and $h \neq 0$ do
        begin if not member($testperm^{-1} * h * testperm$, H)
                then ok:=false;
            h:=next(h);
        end;
        if ok then
        begin Sg(N):=Sg(N)$\cup$[testperm];
          while level>firstmoved do
          begin base_im[level]:=0; level:=level-1
          end;
          calc_N_orbs(level);
        end;
      end;
    end;
  end;
end;
```

We can now list the individual tests that will be applied during the search. Of course, if any of these result in failure, then it is not necessary to continue with the remainder. The tests are arranged roughly in order of decreasing ease of execution. They make use of any variables from the main program that are currently defined, and also some local variables.

Test 1

```
    **if** pointtype[level] **mod** 4=2 **and** testperm[basept] ≠ im_basept **then fail**;
```

REMARK. There are really two different cases involved here, which we can call Test 1a and Test 1b, depending on whether *pointtype* is less than or greater than 8 modulo 16.)

Test 2

```
    if (level=firstmoved and im_basept ≠ N_orb_rep[N_orb_no[im_basept]])
    or (level ≠ firstmoved and H_base_no[level] > 0 and
      im_basept ≠ orbrep[orbim_no][H_orb_no[orbim_no][im_basept]]])
    then fail;
```

Test 3

This attempts to set defim[i] *and* deftime[i] *for* HeqG_no < i ≤ orbim_no, *and it fails if an inconsistency is encountered. Other orbit tests using permutations of the base points, as described in section 3, may also be carried out.*

Test 4

```
  if pointtype[level] mod 4=3 then
  begin hbasept:=H_base[-H_base_no[basept]];
    regstart:=G_base_no[hbasept];
    for i:=regstart+1 to level do if pointtype[i] mod 4=3 then
      compute a permutation $h_i \in H$ mapping hbasept to G_base[i];
    Compute the orbit $ImF_{level}$ of hbasept under the $h_i$.
    if $|ImF_{level}| \neq |F_{level}|$ then fail;
  end;
```

Test 5

```
  if pointtype[level] mod 16=8 then
  begin i:=backpointer[level]; orbno:=imH_orb_no[i][im_basept];
    j:= -H_base_no[basept]; S:=$Sg(H)^{(j+1)}$;
    Check that each permutation of S fixes all points pt satisfying
    imH_orb_no[i][pt]=orbno;
    If any does not then fail;
  end;
```

Test 6

```
  hbno:=H_base_no[basept];
  if hbno>0 then
  begin base_change(H,im_basept,hbno);
```

```
    if hbno<len_H_base then
    begin k:=hbno+1; S:=Sg(H)^(k);
      allorbs(S,imH_orb_no[k],imH_orb_len[k],imH_orb_rep[k]);
      Check that the number and lengths of these orbits correspond to those given
      by H_orb_no[k], etc. If not then fail;
      if pointtype[level] mod 4=1 then
      begin Check that the set ImF^(hbno) which is the intersection of Δ_H^(hbno) with
            the points in orbits of length 1 has the same size as F^(hbno). If not then
            fail;
      end;
    end;
  end;
```

Test 7

```
  if pointtype[level] mod 8≥4 then
  begin i:=backpointer[level]; j:=abs(H_base_no[basept]);
    S:=Old_Sg(H)^(i);
    ok:=true; h:=first(S);
    while ok and h≠0 do
    begin hc:=testperm^-1*h*testperm;
      Use the Schreier vectors for H in the standard fashion to attempt to find and
      store a permutation Imh satisfying hc*Imh^-1∈H^(i+1);
      (c.f. Algorithm "Strip" in (Cannon, 1985).)
      If Inh does not exist then ok:=false;
      h:=next(h);
    end;
    if not ok then fail;
  end;
```

7. Related Algorithms and Possible Improvements

The same algorithm can easily be modified to compute the centralizer of H in G, and the author's implementation has an option to do this. All that we really need to do is to set Img$_i$ equal to g_i for all $g_i \in$ Old_Sg(H), and never change these values. Then all G-base points will have *pointtype* equal to 8 or 10. We store the orbits of $H^{(i)}$ for $i=1$ only. Then the only tests that we need to carry out are Test 1b, and Test 3 with $i=1$. (Test 5 is not applicable in this situation.) This computation nearly always runs very quickly.

The author has also written an algorithm for computing Sylow subgroups of permutation groups, which uses the normalizer algorithm. This is a very different algorithm from that described in Butler & Cannon (1979), which depends on the computation of centralizers, and so it would be interesting to carry out a comparison of their respective performances on a wide variety of examples. What seems to be true is that the author's algorithm finds a Sylow subgroup relatively quickly in some examples (mainly wreath products) on which the Butler & Cannon method is very slow. This does not really provide grounds for a valid comparison, however, since there may well be other examples in which the author's method is slower. We also made a comparison with a soluble group of degree 768 and order $2^{11}3^{12}$, and their performances were similar in this case (about 10 minutes cpu time on a VAX 780 for either prime).

The author's method is to find a p-subgroup H of G and, if this is not already a complete Sylow p-subgroup, a p-element g is sought in $N_G(H)$, and H is replaced by $\langle H, g\rangle$. Initially, the search is fairly simple-minded: the elements of G are considered in order, subject only to the restrictive tests of permuting the orbits of H, and having no cycles of lengths not divisible by p. If no suitable element g is found after a reasonable number of attempts (say 500), then we give up, and use the normalizer algorithm to compute the whole of $N_G(H)$. There remains the problem of finding a p-element in $N_G(H)\backslash H$. (This problem also arises at the beginning, when $H = 1$.) To do this, we look at random elements of $N_G(H)$ until we find an element with order divisible by p, and then replace it by a suitable power, and check that it does not lie in H. If this does not work after a certain number of tries (say 10), then we start trying commutators of the random elements already found; this helps a great deal in certain soluble examples, in which the percentage of p-elements is low. In fact, I have yet to encounter an example in which this technique has not worked fairly quickly.

A further possible application, which has been employed by the author on several occasions, is to use the normalizer algorithm to compute the automorphism group of a group H. The idea is to find a core-free subgroup K of H for which we know in advance that any automorphism of H maps K onto a conjugate of itself. (A Sylow normalizer is a good candidate for such a K.) We then compute the permutation representation of H on the cosets of K, and compute the normalizer of this permutation group in the symmetric group. This will yield all automorphisms of H (and also, of course, the centralizer of H in the symmetric group). This idea was used, for example, to compute Aut(H) when H is the McLaughlin group acting on 275 points. More generally, if K is a core free subgroup of H and H has $t(>1)$ conjugacy classes of (core free) subgroups isomorphic to K, then we can form the intransitive sum of the permutation representations of H on the cosets of a representative of each of these conjugacy classes, and again compute the normalizer in the symmetric group.

We conclude with some performance statistics. In most cases, times are given for the same example run on the CAYLEY group theory system (see Cannon, 1984, for example), for comparison. In this case, x means that the calculation did not complete after running for at least an hour of cpu time (and sometimes much longer). In several cases, CAYLEY did not compute the relevant Sylow-subgroup in a reasonable time, so we had to provide it directly with the generators. The notation for the groups is as follows: Σ denotes the full symmetric group, C_n, E_n and A_n denote cyclic and elementary abelian groups of order n, and the alternating group of degree n, respectively, and Syl_p denotes a Sylow subgroup of G. In this case, the numbers given in brackets are, respectively, the order of the Sylow group and (in some cases) the time taken to compute it—this is relevant, since it also involves runs of the normalizer algorithm on subgroups of the Sylow group. When $G = \Sigma$, we used the CAYLEY function "symmetric normalizer". In several cases, we have cheated slightly by giving the shortest time, after a number of different options (for example, for the choice of the base points for H) had been tried. This does not seem unreasonable for a big calculation, although it does require some intelligent interactive decisions to be taken by the user. All computations were carried out on a SUN 3/60 Workstation, and all times given are in seconds.

These figures must be treated with some caution, since the cpu time recorded for identical runs at different times sometimes varied by up to a 50% difference. Furthermore, for the larger groups like PSL(5,5), the author's program could on occasion complete up to

Degree	H	G	$\lvert N_G(H):H\rvert$	Time	CAYLEY time
11	M_{11}	Σ	1	0·1	3·6
20	PSL(2,19)	Σ	4	0·4	7·0
20	E_{1024}	Σ	$2^9 \times 10!$	0·8	?
24	M_{24}	Σ	1	7·8	54·6
32	E_{32}(regular)	Σ	$2^{10} \times 3^2 \times 5 \times 7 \times 31$	0·3	33·2
32	AGL(5,2)	Σ	1	134	98·8
49	AGL(2,7)	Σ	1	573	24·3
50	PSL(2,49)	Σ	4	4·0	95·8
50	PSU(3,5)	Σ	2	1·4	x
60	A_5(regular)	Σ	120	0·7	x
64	C_{64}	Σ	32	5·6	34·6
64	E_{64}(regular)	Σ	$2^{15} \times 3^4 \times 5 \times 7^2 \times 31$	8·1	2252
97	C_{97}	Σ	96	11·4	93·5
97	D_{194}	Σ	48	155	130
100	J_2	Σ	2	4·3	184
100	HS	Σ	2	56·4	x
121	$Syl_2(2^9,0{\cdot}8)$	PSL(5,3)	1	4·0	6·0
121	$Syl_5(5,1{\cdot}3)$	PSL(5,3)	16	3·1	5·1
121	$Syl_{11}(11^2,1{\cdot}5)$	PSL(5,3)	5	4·5	x
210	$Syl_2(2^{17},115)$	A_{21}	1	17·4	x
210	$Syl_7(7^3,65{\cdot}7)$	A_{21}	$2^3 \times 3^4$	9·1	x
210	A_7	A_{21}	1	11·1	x
275	McL	Σ	2	266	x
275	$Syl_2(2^7)$	McL	1	6·4	7·5
275	$Syl_{11}(11)$	McL	5	4·9	30·7
364	$Syl_2(2^{11},152)$	PSL(6,3)	1	27·9	36·9
364	$Syl_3(3^{15},8{\cdot}5)$	PSL(6,3)	16	54·2	34·1
364	$Syl_5(5,3{\cdot}5)$	PSL(6,3)	$2^8 \times 3$	27·9	99·6
364	$Syl_{11}(11^2,4{\cdot}2)$	PSL(6,3)	5	34·0	x
364	C_4	PSL(6,3)	$2^9 \times 3^2$	30·6	31·5
781	$Syl_2(2^{11},74{\cdot}7)$	PSL(5,5)	1	82·8	53·3
781	$Syl_3(3^2,232)$	PSL(5,5)	2^9	100	142
781	$Syl_5(5^{10},29{\cdot}9)$	PSL(5,5)	2^8	293	70·2
781	$Syl_{13}(13,5{\cdot}1)$	PSL(5,5)	$2^6 \times 3$	70·9	589
781	$Syl_{71}(71,2{\cdot}6)$	PSL(5,5)	55	96·0	x
781	C_3	PSL(5,5)	$2^9 \times 3 \times 5^3 \times 31$	157	159
2709	$Syl_2(2^{21},179)$	PSU(7,2)	27	172	116

10 times quicker by increasing the available storage, since coset representatives in the stabilizer chain for G could then be stored as permutations rather than indirectly, using Schreier vectors. Another consideration is that the CAYLEY times may be slowed down by a factor of 2 or 3 by the overheads present in CAYLEY's handling of data. The performances seem to suggest that the algorithm represents a significant improvement over existing algorithms for some classes of groups (particularly when G is the full symmetric group and when H has large or moderately large orbits on which it acts regularly), and to be about the same as existing algorithms in most other cases. It cannot be denied, however, that there a few cases (such as the AGL groups) in which it is not doing very well, and so there is still room for further refinements. Since I am not familiar with the details of the algorithms currently used in CAYLEY, I am not able to explain the reason why the CAYLEY times are so variable; in particular, I do not understand its relatively good performance on examples like H = AGL(2,7), or indeed its poor performance on H = PSU(3,5).

References

Butler, G. (1982). Computing in permutation and matrix groups II: backtrack algorithm. *Math. Comp.* **39,** 671–680.

Butler, G. (1983). Computing normalizers in permutation groups. *J. Algorithms* **4,** 163–175.

Butler, G., Cannon, J. (1989). Computing in permutation and matrix groups III: Sylow subgroups. *J. Symbolic Computation* **8,** 241–252.

Cannon, J. (1984). An introduction to the group theory language Cayley. In: (Atkinson, M., ed.). *Computational Group Theory, Proc. of LMS Meeting* (Durham, 1982), pp. 145–183.

Cannon, J. (1985). A computational toolkit for finite permutation groups. In: *Proc. Rutgers Group Theory Year, 1983–84,* (New Brunswick, NJ, 1983–84). New York: Cambridge University Press, pp. 1–18.

Leon, J. (1984). Computing automorphism groups of combinatorial objects. In: (Atkinson, M., ed.) *Computational Group Theory, Proc. LMS Meeting* (Durham, 1982), pp. 321–335.

Sims, C. C. (1971*a*). Determining the conjugacy classes of a permutation group. In: (Birkhoff, G. and Hall, M., eds). *Proc. Symp. on Applied Mathematics* (New York, 1970), *SIAM–AMS Proc.* **4,** Providence, RI: Amer. Math. Soc., pp. 191–195.

Sims, C. C. (1971*b*). Computation with permutation groups. In: (S. R. Petrick, ed.). Proc. Second Symp. on Symbolic and Algebraic Manipulation (Los Angeles, 1971). New York: Association of Computing Machinery, pp. 23–28.