A Data Structure for a Uniform Approach to Computations with Finite Groups

Max Neunhöffer
Lehrstuhl D für Mathematik
RWTH Aachen
Templergraben 64
D-52062 Aachen, Germany
max.neunhoeffer@math.rwth-aachen.de

Ákos Seress
The Ohio State University
Department of Mathematics
231 W. 18th Avenue
Columbus, Ohio 43210, USA
akos@math.ohio-state.edu

ABSTRACT

We describe a recursive data structure for the uniform handling of permutation groups and matrix groups. This data structure allows the switching between permutation and matrix representations of segments of the input group, and has wide-ranging applications. It provides a framework to process theoretical algorithms which were considered too complicated for implementation such as the asymptotically fastest algorithms for the basic handling of large-base permutation groups and for Sylow subgroup computations in arbitrary permutation groups. It also facilitates the basic handling of matrix groups. The data structure is general enough for the easy incorporation of any matrix group or permutation group algorithm code; in particular, the library functions of the *GAP* computer algebra system dealing with permutation groups and matrix groups work with a minimal modification.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; E.1 [**Data structures**]: Records; I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Algorithms*

General Terms

Algorithms

Keywords

Computational group theory, permutation group algorithm, matrix group algorithm, black-box group, constructive membership, large-base group, *GAP*

1. INTRODUCTION

There are two basic methods to input a group into a computer: (a) by a presentation, using abstract generators and relations, and (b) by a "concrete" representation as a permutation group or matrix group, defined by a set of generating permutations or matrices. The major difference between these two ways is that in a group given by a presentation, by the celebrated result of Novikov and Boone [15, Ch. 12], it is recursively undecidable whether a string represents the identity element of the group; on the other hand, for any permutation or matrix, we can decide whether it is the identity. In this paper, we are concerned with groups given as in (b), and with black-box groups, which are a common generalization of permutation and matrix groups.

Definition 1. A *black-box group* is a group whose elements are encoded as 0-1 strings of uniform length N, and an oracle (the "black box") performs the following three operations: given $g, h \in G$, it can compute (a string representing) gh, g^{-1}, and can decide whether $g = 1$.

Given $G = \langle X \rangle$, a *straight-line program* (*SLP*) *of length* m, reaching some element $g \in G$ from X, is a sequence of expressions $(w_1, \ldots, w_m)$ such that for each i one of the following holds:

- w_i is a symbol for some element of X,
- $w_i = (j, -1)$ for some j with $0 < j < i$, or
- $w_i = (j, k)$ for some j, k with $0 < j, k < i$,

such that if the expressions are evaluated then the value of w_m is g. Here, if w_i is a symbol for some element of X then it is evaluated as that element, $(j, -1)$ is evaluated as the inverse of the evaluated value of w_j, and (j, k) is evaluated as the product of the evaluated values of w_j and w_k. Straight-line programs can be considered as shortcuts for long words in the generators. We shall also use the expression that an SLP reaches a set of group elements from X if the evaluated values of some subsequence of the w_i consist of these group elements.

For the basic manipulation of black-box groups, there are two main types of algorithms: constructive membership and reduction.

Definition 2. We say that an algorithm *solves the constructive membership problem in a black-box group* $G = \langle X \rangle$ if it computes $|G|$, constructs a new generating set Y from X by a sequence of steps multiplying and inverting group elements, and sets up a procedure that, for any given $g \in G$, computes a straight-line program from Y to g.

We say that an algorithm is a *reduction algorithm for a black-box group* G if it defines a homomorphism $\varphi : G \rightarrow$

H for some group H with nontrivial image, and sets up a procedure that, for any given $g \in G$, computes $\varphi(g)$.

For permutation groups G, the Schreier–Sims algorithm and its variants [17],[16] solve the constructive membership problem in G, by computing a strong generating set Y. However, for large-base permutation groups (see Section 3), there are asymptotically more efficient algorithms, which use reduction first. Also, it does not seem possible to solve the constructive membership problem in arbitrary matrix groups directly; all algorithmic attempts go through a series of reduction steps, until reaching a permutation group, or an almost simple matrix group modulo scalars, or a solvable matrix group. Another need arises for reduction algorithms when we go beyond the basic task of computing the order of the input permutation or matrix group G, and set up membership testing in G: for example, if we want to compute a composition series or Sylow subgroups.

The purpose of this paper is to describe a data structure which provides a framework for the search for applicable reduction algorithms and then, if a suitable reduction algorithm is found, for the division of the computations in the image and kernel of this reduction. The data structure is a rooted tree, with the root corresponding to the input G, and other nodes corresponding to certain *subnormal segments* of G. (Recall that the *segments* of G are the subgroups and quotients of subgroups of G, and a subgroup $H < G$ is *subnormal* in G if there is a series of subgroups $H = H_0 < H_1 < \cdots < H_m = G$ such that $H_i \lhd H_{i+1}$ for all $i < m$. We call a segment subnormal if the subgroup in its definition is subnormal.) The inner nodes K have a reduction algorithm associated with them, and they have two children: the image and the kernel of a homomorphism from K. The leaves have an algorithm for constructive membership associated with them.

The data structure is implemented in the *GAP* computer algebra system [11], taking full advantage of the object-oriented features of *GAP* and using ideas of the method selection mechanism of *GAP* for the choice of reduction methods. The data structure is implemented on the black-box group level, so it is equally suitable for matrix groups and permutation groups. In most applications, the type of change (if there is a change at all) is from matrix groups to permutation groups but, for example, the asymptotically fastest Sylow subgroup computations in permutation groups require to switch to matrix representations of certain composition factors. There are also applications of constructive membership in simple groups which switch from matrix or permutation groups to black-box groups.

In the permutation group setting, the use of homomorphic images to break a computation into manageable pieces was introduced by Luks, in the context of parallel (NC) computations [14],[6]. As we mentioned earlier, in the matrix group setting this seems to be the only feasible approach. Concerning implementations, there exist data structures similar to ours; the most notable one is described in [13] and is implemented in MAGMA [9] by Eamonn O'Brien for computations with matrix groups. Compared with previous *GAP* data structures for similar purposes like `HomCoset`, our approach has the advantage that it is compatible with the permutation group and matrix group algorithms of the GAP library. Also, there are numerous features that go beyond all such previous efforts.

One novelty of our approach is the generality of application to all black-box groups. In fact, our first success story is the uniform handling of all permutation groups (i.e., both small-base and large-base groups, see Section 3.1 for details). There are also many new ideas to improve efficiency; here we highlight only two of them.

A mechanism is developed so that a node V in our recursive tree structure can give hints to its descendants, suggesting methods for a possible reduction algorithm. This way, information gathered by the node V, which may be lost for example if a child of V gets only generators for the kernel of the reduction homomorphism of V, can be passed down. These hints also enable each node to have an individualized method selection process, which may find a suitable reduction algorithm faster than trying all applicable methods in a uniformly prescribed order.

The second idea we would like to highlight here is that we construct a new generating set such that group elements can be reached by much shorter straight-line programs than from the input generators. In the permutation group setting the traditional strong generating sets achieved this goal, but in the matrix group setting previous attempts used straight-line programs from the input generators.

Our setup also makes possible a clean design of algorithms with complicated case analysis. For example, Alice Niemeyer used our method selection process to implement an algorithm for the non-constructive recognition (i.e., finding the name of the isomorphism type) of classical almost simple matrix groups, in their natural representation. Our data structure was carefully designed to accommodate subroutines written by other developers, who do not have to be aware of the syntax or any detail of our implementation. Already there are numerous examples of incorporating algorithms written independently in our framework. Given the decentralized nature of *GAP* development, such an approach is essential for cooperation with other developers of permutation or matrix group code. Part of our code is available as a formal *GAP* package called `recog`, while other parts are incorporated in the *GAP* library.

2. BUILDING THE DATA STRUCTURE

Our data structure to handle a group $G = \langle X \rangle$ is a rooted tree, with the nodes corresponding to a *recognition info* record for certain subgroups of G. The nodes of the tree are created by a highly recursive procedure. The input at the creation of a new node V is a group K, given by a list X_K of generators. The group K is a segment of G. The goal is to set up a procedure which, given any $k \in K$, writes a straight-line program reaching k from X_K. This can be achieved by either finding an algorithm for constructive membership in K, and in this case the node V becomes a leaf of the tree, or by finding a reduction algorithm, that is, finding a homomorphism $\varphi : K \to H$ onto some group H. In the latter case, we solve the problem of writing the elements of $\mathrm{Im}(\varphi)$ and $\mathrm{Ker}(\varphi)$ in terms of generators of these groups recursively, and put together a procedure for the elements of K from these two subprocedures (see Subsections 2.1 and 2.4).

The entire algorithm starts with the creation of the root by inputting the generators X for G, and the algorithm terminates when the recognition info record of the root is complete.

2.1 Tasks of a node

To make our recursive recognition procedure work, each node in the tree — be it a leaf or not — has to fulfill the following tasks for its input group K with generators X_K:

(1) Recognize the group by finding possibly new "nice generators" Y_K and specifying an algorithm to solve the constructive membership problem, that is, given any group element $k \in K$, being able to return an SLP reaching k from the nice generators Y_K.
(2) Record how the nice generators Y_K were derived from the input generating set X_K. That means, that the node has to be able to calculate, given preimages of the X_K under some group homomorphism $\theta : L \to K$, preimages of the Y_K under θ.

Note that for (1) it is of course possible that $Y_K = X_K$. Task (2) can be achieved by recording an SLP that reaches Y_K from X_K, because this SLP can be executed with the given preimages of the X_K as input thereby reaching preimages of the Y_K.

For leaf nodes, these two tasks amount to exactly what we have formulated in Definition 2 plus recording how Y_K was obtained from X_K.

For a non-leaf node, we can fulfill the two tasks by recursively using the results of the recognition for the image and the kernel of the homomorphism $\varphi : K \to H$ in the following way:

The set of new nice generators Y_K of a non-leaf node is the union of some preimages Y_H^K of the nice generators Y_H of H under φ and the set Y_N of nice generators of the kernel N of φ. We recursively ask the node of H to calculate $Y_H^K \subseteq K$. We call task (2) of that node with input X_K, which we can do, because the input generators X_H of that node are by definition the images of X_K under φ. The input generators X_N of the kernel were produced during the recognition of the node for K and after the recognition of H, therefore we can just store how we got them from X_K and Y_H^K by storing an SLP.

If we are given any element k of K we can find an SLP that reaches it from $Y_K = Y_H^K \cup Y_N$ in the following way: We first map k via the homomorphism φ into H and recursively find an SLP reaching $\varphi(k)$ from the Y_H in H. Applying this SLP with inputs Y_H^K reaches an element $k' \in K$ which is mapped to the same image under φ as k, such that kk'^{-1} lies in N. We then ask the node for N recursively for an SLP reaching kk'^{-1} from Y_N. We can then combine those two SLPs into one SLP reaching k from $Y_K = Y_H^K \cup Y_N$. Thus we can fulfill task (1) in non-leaf nodes.

As to task (2), if some procedure further up in the tree gives us preimages X_L of the X_K under some homomorphism $\theta : L \to K$, we first ask the node for H to calculate preimages Y_H^L of the Y_H^K under θ by giving it the preimages X_L, which are preimages of X_H under $\varphi \circ \theta$. Then we execute the stored SLP with inputs X_L and Y_H^L to find preimages X_N^L of the input generators X_N of N under θ and finally call the node for N recursively to find preimages Y_N^L of the Y_K under θ. Then Y_H^L and Y_N^L are preimages of the nice generators for K under θ. Thus we also can fulfill task (2) for a non-leaf node.

This shows that both tasks can be fulfilled in a non-leaf node.

2.2 Method selection for processing a node

After a new node V is created by specifying a group $K = \langle X_K \rangle$, we have to explore K to find out whether we can solve the constructive membership problem, or what kind of homomorphism can be applied to it. To this end, the framework holds a collection of so-called "find homomorphism" methods in stock. A find homomorphism method's objective is either to find a homomorphism $\varphi : K \to H$ onto some subgroup H, thereby creating a new non-leaf node, or to solve the constructive membership problem directly, which can but does not always have to find an isomorphism to some known group.

For each type of groups (permutation groups, matrix groups, and black-box groups), the system has a database of find homomorphism methods. We call the procedure that decides, which methods to try and in which order the "method selection". For this purpose we define a very simple and yet versatile algorithm, which we will describe now. It is usable independently from group recognition, but we will explain it here in the context of our generic recognition procedure. Note that this new method selection procedure is not to be confused with the GAP method selection.

The group recognition procedure for a node just calls the generic method selection procedure with the database of find homomorphism methods corresponding to the type of the group.

The methods in each database are ranked, thereby defining a total order. The method selection procedure calls the methods one after another, starting with high ranks. A find homomorphism method reports back to the generic procedure by returning one of the following four values:

`true`:
The method was successful and has either set up a leaf or a non-leaf node. For details see below.

`fail`:
The method has failed to find a homomorphism or to solve the constructive membership problem, at least temporarily.

`false`:
The method has failed and will do so always for the group K in question, such that there is no point in trying this method again for the group K.

`NotApplicable`:
The method is currently not applicable but it might become applicable later, provided new knowledge is found out about the group K.

The first case is the only one that terminates the recognition procedure for the current node and we will discuss this case below in detail. All other cases make it necessary to try other methods. The difference between these latter three cases lies in the fact, how the generic procedure chooses the next method called. If a method returns `NotApplicable`, then the method selection just calls the next method in the database. In the other two cases `false` and `fail`, the method selection again starts with the highest ranked method, but skipping all methods that have already been tried and have failed by returning either `false` or `fail`.

When all available methods either have declared themselves `NotApplicable` or have failed, then the method selection starts all over again, now calling methods again that have returned `fail` once but of course skipping methods that have returned `false`. This whole process is repeated until each method has failed a configurable number of times, when the method selection finally gives up.

We hope that this design is simple enough to keep an overview of what is tried in which order to prove the whole algorithm to work correctly, and versatile enough to implement a wide range of different algorithms, in which "trying different methods" is involved. We now explain the rationale behind some of the features of this procedure.

The idea behind the fact that after a method having returned `fail` or `false` the method selection starts again with the highest ranked method is that even a failed method might have found out new information about the group, thereby making higher ranked methods, that have refused to work earlier by returning `NotApplicable`, newly applicable.

Information about K is collected in so-called *attributes* (for example, a permutation group object may store an attribute whether it is transitive or not), but we may also acquire the information that K is simple, or that it is solvable, or we may know $|K|$, etc. Note that at any given time, the value of an attribute for a given group object can already be computed or not, and the group object can learn new information about itself during its lifetime. This feature is already part of the GAP system library.

Further attributes may be computed when the method selection tries to apply different find homomorphism methods while processing the node V. Therefore, a find homomorphism method can just look whether or not a certain attribute is already known and then decide if it starts to work or declares itself `NotApplicable`. With new attributes being computed, this decision might be changed. By convention, a method should never use much computation time to find out that it is not applicable.

The idea behind a method returning `fail` is that often randomized algorithms are used that have the potential to fail, but still may succeed when tried again. The ranking and the failure probabilities of course have to be tuned carefully to assemble a sensible recognition system.

We give a few examples to explain the procedure: For permutation groups there is a very high ranked method called "Intransitive" for non-transitive groups that finds a homomorphism by restricting to an orbit. If called, it computes whether the group is transitive. If so, it has succeeded to find a homomorphism and returns `true`. If the group is transitive, it returns `false`, because this method will never find a homomorphism.

Another method "Imprimitive" tries to find a homomorphism by finding a block system. This method first checks whether the group is known to be transitive (without triggering a computation). If not, it returns `NotApplicable`. If the group is known to be transitive, it starts computing a block system using standard GAP library methods. If one is found, the method succeeded to find a homomorphism and returns `true`, otherwise it returns `false`, because this method will not work, even when tried again. Note that the "Intransitive" and the "Imprimitive" methods correctly work together regardless of their respective ranking in the database. The system will first try to restrict to an orbit and only if this is not possible search for a block system.

A third type of behaviour is shown by a randomized method to recognize a "giant", which means that the group is either an alternating or symmetric group in its natural action. This method tries for some time to find elements whose existence proves that the input is a giant, and if not successful returns `fail`. Due to the probabilistic nature of the algorithm the method could finally succeed when called again.

We now explain how a successful method reports back to the generic recognition procedure what it has found. It does so by setting certain attributes of the recognition info record.

A method returning `true` has to either report back a homomorphism in form of a GAP homomorphism object, by which the system can map elements through the homomorphism. Or, for leaf nodes, the method has to declare the node to be a leaf and also return information about its chosen nice generators and the guaranteed procedure to solve the constructive membership problem for this node. A variety of other reporting possibilities are provided in the framework by attributes of recognition info records. See Subsection 2.6 for details about this.

2.3 Group elements with memory

In the case of leaves, when we solve the constructive membership problem in the group $K = \langle X_K \rangle$ associated with the leaf node V, we usually compute a new generating set Y_K for K which is suitable to write straight-line programs from Y_K reaching any element of K. However, for the recursive recognition procedure outlined in Subsection 2.1, we have to record how Y_K was derived from X_K. The easiest approach for this is that the construction of Y_K records an SLP from X_K to Y_K Another instance for the necessity of recording an SLP at the construction of group elements arises at inner nodes of the tree structure. When a homomorphism $\varphi : K \to H$ is created from the group $K = \langle X_K \rangle$ associated with the node, we need SLP's from $X_K \cup Y_H^K$ reaching the generators X_N of $\mathrm{Ker}(\varphi)$.

The need of recording the steps at the construction of group elements gives rise to two conflicting requirements. On one hand, every group operation has to be recorded in an SLP; on the other hand, this recording should not be part of the code of the applied methods, since *we want to use existing permutation and matrix group algorithms without rewriting the library of GAP functions*. The solution is the definition of an SLP S which records every group operation we do starting with a certain set M of generators, and the introduction of new objects "permutation with memory" and "matrix with memory". The SLP S is initialized to contain a list of symbols for the elements of M. The new data types are records (wrapped up in the object mechanism of *GAP*), consisting of a permutation or matrix g, a pointer to S, and a number, indicating the position of g in S. We install new methods for the multiplication, inversion, powering, conjugation, and comparison of these new objects: these methods perform the appropriate operations on the permutation or matrix part of the objects, add the operation performed to S, and store the new length of S in the resulting object (since this length is the position of the result in S). We note that SLP's in *GAP* are slightly more general than the ones described in Definition 1, allowing for example the addition of an arbitrary power of a previous element, or an arbitrarily long product of previously defined elements in one step.

Any existing (and future) permutation or matrix group algorithm, which creates new permutations and matrices only by black-box operations (i.e., multiplication and inversion of previously constructed elements), works with these new data objects without any modification. There are only a

handful of instances in the current *GAP* library where this condition is not satisfied, and it is possible to work around those situations without too much effort.

The very same implementation can be used to describe words in a finitely generated free group by storing those words as a reference in the (universal) straight-line program S that produces all elements that occurred in a certain calculation. This approach is very similar to the concept of "SLPElements" available in the MAGMA [9] system.

Note that our approach to let each node choose a "nicer generating set" and to always write SLPs in terms of those has two major advantages: firstly our SLPs will be much shorter and secondly the memory for group elements only has to be used "locally" within each node and not all across the recognition tree.

2.4 Recursion

When processing a node V with corresponding group $K = \langle X_K \rangle$, any homomorphism $\varphi : K \to H$ created by a find homomorphism method must satisfy the property that there exists a function f which, for all $g \in K$, computes $\varphi(g)$. This function usually requires some auxiliary data structure, which must be stored in the recognition info record V.

For example, if $K \leq \mathrm{Sym}(n)$ is an intransitive permutation group and φ is the restriction to an orbit O of length l then the auxiliary data is a permutation $\pi \in \mathrm{Sym}(n)$ mapping O to $\{1, 2, \ldots, l\}$, and the number l. The function f is conjugation by π, followed by the restriction to $\{1, 2, \ldots, l\}$. Of course, for other homomorphisms the auxiliary data and the function f may be much more complicated. We defined a new generic operation in *GAP*, `GroupHomomorphismByFunctionWithData`, to accommodate such homomorphisms.

If a find homomorphism method succeeds to find φ then it creates a new node $R(V)$ (the right-hand-child of V) with associated group $H := \mathrm{Im}(\varphi)$. The input generator set X_H for H is by definition equal to $\varphi(X_K)$, but of course as explained in Subsection 2.1, the child node may choose another nice generating set Y_H. Then we recursively process $R(V)$, until a data structure is set up to write an SLP to any $h \in H$ from Y_H. The next step is to store preimages Y_H^K of the nice generators Y_H under φ within the node V. This can be done by using task (2) described in Subsection 2.1 for the node $R(V)$. After that, we define $L(V)$ (the left-hand-child of V), which corresponds to $\mathrm{Ker}(\varphi)$.

There are two basic methods to construct a generating set X_N for $N := \mathrm{Ker}(\varphi)$. The generic one is to construct random elements $k \in K$, compute $\varphi(k)$, write an SLP S from Y_H reaching $\varphi(k)$ in H using the recognition info record $R(V)$, and evaluate S using the stored preimages Y_H^K of Y_H in K. The result of this evaluation is some $k' \in K$, and then kk'^{-1} is an element of N. Each element of N has the same probability to occur as kk'^{-1}, hence repeating this procedure enough times we obtain a generating set for N with arbitrarily high probability.

The second method for obtaining generators for $N = \mathrm{Ker}(\varphi)$ is applicable if in H the constructive membership problem is solved using the nice generating set Y_H *and* we also have a presentation $\langle Y_H \mid r_1, \ldots, r_m \rangle$ with some relators $r_1, \ldots, r_m$. (During the construction of our data structure, this situation usually, but not exclusively, occurs if H is almost simple.) Then the following set `gensN` is a set of *normal generators* for N, i.e., the normal closure $\langle \texttt{gensN}^K \rangle = N$. The set `gensN` is constructed in the following way: For each $x \in X_K$, compute $\varphi(x)$; write an SLP S from Y_H to $\varphi(x)$; compute the evaluation y of S using preimages Y_H^K of Y_H in K; add xy^{-1} to `gensN`. Also evaluate the relators $r_1, \ldots, r_m$ using the preimages Y_H^K of Y_H and add the evaluated values to `gensN`. After `gensN` is constructed, its normal closure can be obtained by the black-box methods of [10] or [3] (see also [16, Ch. 2]).

The attribute `findgensNmethod` records the method used to construct generators of the kernel $N = \mathrm{Ker}(\varphi)$ and can be set by the successful find homomorphism method. Note that during the construction of generators for N we have to "remember" how the generators are obtained from X_K and Y_H^K (again see Subsection 2.1). After generators for N are constructed, we recursively process $L(V)$.

The generic function for writing SLP's reaching given elements of K from Y_K is `SLPforElementGeneric`. It does what is described near the end of Subsection 2.1 to fulfill task (1) for homomorphism nodes.

2.5 Verification

The construction of our data structure is randomized. It is a *Monte Carlo* procedure, which means that the output may be incorrect; however, the user can prescribe an upper bound for the probability of erroneous output. Hence, after the construction of the data structure, we may have a verification phase which proves the correctness of the output. The verification is done by constructing and evaluating a presentation for the input group G. The presentation is constructed by a recursive procedure in a bottom-up manner, starting at the leaves.

Presentations for the groups K associated with leaves may be constructed by diverse methods, depending on the type of K. At inner nodes of the tree, we employ a generic procedure from [5], see also [16, Sect. 8.4], which, given presentations for N and K/N for some group K and $N \lhd K$, and procedures to write SLP's in N and K/N from the generators to any element of N and K/N, respectively, constructs a presentation for K.

The verification procedure is not yet implemented in *GAP*.

2.6 Components of the recognition info record

In this subsection we summarize the components of the recognition info record (or "component object", as it is called officially in *GAP*). The components are filters and attributes. A *filter* is a class of objects the recognition info may or may not belong to, indicated by a flag stored in the type of the object. An *attribute* is some information about the recognition info record together with the information whether this attribute has already been determined or not. All attributes start with being not yet determined, then most are initialized with sensible default values by the generic recursive recognition procedure. Find homomorphism methods (especially successful ones) later can change or set the values to report information back to the generic procedure.

Filters:

`IsLeaf`: Indicates whether we are on a leaf in the tree. This has to be set by a successful find homomorphism method.

`IsReady`: Indicates whether the node is already set up completely, which means that the recognition was successful.

`DoNotRecurse`: Tells the generic procedure that the node might or might not be a leaf. However, the find homomorphism takes complete responsibility to set it and all its children up.

Generic attributes: (always known in the end)
`group`: The group object $K = \langle X_K \rangle$ at this position of the tree.
`nicegens`: The list Y_K of nice generators.
`parent`: A possible parent node in the tree, only the top node has none set.
`fhmethsel`: A GAP record that keeps record of the method selection process at this node. For example one can see which methods have been tried how often and which one succeeded.
`slpforelement`: A GAP function fulfilling task (1) of the node, which returns, given two arguments `ri` and `x`, an SLP reaching the group element `x` of K from Y_K (`ri` must be the recognition info record of this node).
`presentation`: This attribute is reserved for the future implementation of the verification phase.
`calcnicegens`: A GAP function fulfilling task (2) of the node, which returns, given preimages of the input generators X_K under some homomorphism $\theta : L \to K$, preimages of the nice generators Y_K under θ. The default function is to take the value of the attribute `slptonice`, which has to contain (if set) an SLP reaching Y_K from X_K.
`slptonice`: See the previous attribute.
Generic attributes for the non-leaf case (and maybe in some leaf-cases):
`homom`: A GAP homomorphism object describing $\varphi : K \to H$ for some group H.
`pregensfac`: The set Y_H^K of preimages of the nice generators in the image.
`factor`: The right-hand-side child corresponding to the image group H in the recognition tree. Its value is again a recognition info record.
`kernel`: The left-hand-side child corresponding to the kernel group N in the recognition tree or `fail` if the kernel is known to be trivial.
`gensN`: A list collecting generators of N.
`gensNslp`: An SLP to write generators of N in terms of X_K and Y_H^K.
`findgensNmethod`: A record with components `method` and `arg`. The function `method` is called with argument list `arg` (with the recognition info record prepended) to create the generators of the kernel.
`methodsforfactor`: The database used for recognizing the factor group H. This for example has to be changed by a find homomorphism method mapping from a matrix group onto a permutation group.
`immediateverification`: A flag (default value `false`) that indicates if an additional (randomized) verification stage is done after completing the recognition of the kernel (namely constructing a few more kernel elements and trying to find an SLP reaching them). This is important if a find homomorphism method learns that the kernel N will need lots of generators.
`forfactor`: A record where find homomorphism methods can store information that will be available during the recognition of the factor group H. In particular, the component `hints` can be used to store additional find homomorphism methods that will be tried first when recognizing the factor.
`forkernel`: A record where find homomorphism methods can store information that will be available during the recognition of the kernel N. In particular, the component `hints` can be used to store additional find homomorphism methods that will be tried first when recognizing the kernel N.

Additional data: depending on the type of homomorphism found.

2.7 Two examples

We give a small example of the installation of find homomorphism methods that actually finds a homomorphism. The following code tries to restrict the action of an intransitive permutation group to one of its orbits:

```
FindHomMethodsPerm.NonTransitive :=
   function( ri, G )
        local hom,la,o;
        # Test whether we can do something:
        if IsTransitive(G) then
            return false; # do not call us again
        fi;
        la := LargestMovedPoint(G);
        o := Orbit(G,la);
        hom := ActionHomomorphism(G,o);
        Sethomom(ri,hom);
        return true;
   end;
AddMethod(FindHomDbPerm,
   FindHomMethodsPerm.NonTransitive,
   90, "NonTransitive");
```

Our second small example shows the installation of a find homomorphism method that creates a leaf. The following code recognizes the trivial permutation group and sets up a corresponding leaf:

```
SLPforElementFuncsPerm.TrivialPermGroup :=
   function(ri,g)
        return StraightLineProgram( [ [1,0] ], 1 );
   end;
FindHomMethodsPerm.TrivialPermGroup :=
   function(ri, G)
   local g,gens;
   gens := GeneratorsOfGroup(G);
   for g in gens do
        if not(IsOne(g)) then
            return false;
        fi;
   od;
   Setslpforelement(ri,
        SLPforElementFuncsPerm.TrivialPermGroup);
   Setslptonice( ri,
        StraightLineProgram([[[1,0]]],
        Length(GeneratorsOfGroup(G))));
   SetFilterObj(ri,IsLeaf);
   return true;
end;
AddMethod(FindHomDbPerm,
   FindHomMethodsPerm.TrivialPermGroup,
   110, "TrivialPermGroup");
```

This code refers to the also shown `SLPforElement` function which constructs an SLP for the elements of K. Although this example is rather trivial, it shows the mechanisms of the recognition framework.

3. APPLICATIONS

3.1 Permutation groups

A *base* for a permutation group $G \leq \mathrm{Sym}(\Omega)$, with $|\Omega| = n$, is a sequence $B = (\beta_1, \ldots, \beta_m)$ of points from Ω such that the pointwise stabilizer of B in G is the identity subgroup. The minimal base size of G and $\log |G|$ are within a factor $\log n$ of each other. A base defines a series of subgroups $G = G^{[1]} \geq G^{[2]} \geq \cdots \geq G^{[m+1]} = 1$, where $G^{[i]}$ is the pointwise stabilizer of the first $i-1$ base points. A *strong generating set* (SGS) relative to B is a generating set S for G such that $S \cap G^{[i]}$ generates $G^{[i]}$ for all $i \leq m$. The Schreier–Sims algorithm [17] constructs a base and SGS for permutation groups G. This data structure allows us to compute $|G|$ and test membership in G.

Given $G = \langle X \rangle \leq \mathrm{Sym}(n)$, randomized versions of the Schreier–Sims algorithm [4], [16, Ch. 4] compute a base and SGS in $O(n|X|\log^c |G|)$ time, for some small constant c. In the family of *small-base groups*, where $\log |G|$ (and so the minimal base size) is bounded from above by a polylogarithmic function of the permutation degree, such algorithms run in *nearly linear*, $O(n|X|\log^{c'}(n|X|))$, time of the input length $n|X|$. The current implementation for the basic handling of permutation groups in *GAP* is such a nearly linear-time algorithm.

By definition, a family of permutation groups is called *large-base* if it is not in the family of small-base groups. For large-base groups, the running times of the algorithms in [4], [16, Ch. 4] are not nearly linear, and there is a deterministic algorithm [7] which computes $|G|$ and tests membership in G asymptotically faster than any version of the Schreier–Sims approach. The basic idea behind the algorithm in [7] is to detect the large alternating composition factors of the input, apply special algorithms for those, and handle the rest of the group by the traditional Schreier–Sims technique. The algorithm of [7] has not been implemented earlier.

Our new data structure enables the development of a *unified* algorithm for all permutation groups, which handles both small-base and large-base groups. We installed find homomorphism methods for orbit and imprimitivity block calculation, and for handling symmetric and alternating groups in their natural representation (by speeding up a black-box recognition algorithm for alternating groups [8], along the lines described in [16, Sect. 10.2.4]; this last program was written by Maska Law). We also incorporated a randomized version [12] of the algorithms in [7, Sect. 4], to obtain a find homomorphism method for the handling of primitive permutation groups with large alternating groups "hidden" in their socle. The rest of the algorithm of [7] is done automatically, by the generic functions of the new framework, with one noteworthy addition. In the case when a find homomorphism method finds a block system $\mathcal{B}$ and the homomorphism $\varphi : K \to \mathrm{Sym}(\mathcal{B})$ in the group K associated with a node V, then this information is passed down to $L(V)$. The node $L(V)$ tests whether the action of $\mathrm{Ker}(\varphi)$ on one block is solvable. If the answer is yes then $\mathrm{Ker}(\varphi)$ is solvable, $L(V)$ becomes a leaf node, and it is handled by Sims's algorithm for solvable permutation groups [18]. If the answer is no then the intransitive methods take over, eventually creating $|\mathcal{B}|$ homomorphisms onto the blocks. We instruct $L(V)$ and all of its descendants to divide the blocks they handle into two equal groups, and create a homomorphism as a restriction to one of these halves. This way, we can achieve that the subtree below $L(V)$ is balanced, which speeds up the recursive calls significantly.

Experiments show that for permutation degree $n > 100$, the unified algorithm is about as fast as the nearly linear-time Schreier–Sims algorithm in the case when the input is a small-base group, and it performs considerably faster than the Schreier–Sims algorithm in the case of large-base inputs.

The following table contains some runtimes. We compare our recursive recognition approach with the method database for permutation groups with the randomized calculation of a stabilizer chain (error probability 10%), both for the case that the group order is known beforehand (indicated by "o.k.") and not. Note that our approach cannot benefit from knowledge of the group order, whereas the stabilizer chain verification phase can. All times are in seconds and were performed on a machine with a Pentium 4 processor running at 2.5 GHz with 1.5 GB of main memory.

G	Rec.	StabChain (o.k.)	StabChain
$S_2 \wr S_{100}$	2.0s	10.5s	78.8s
$A_5 \wr S_{100}$	7.2s	124.0s	1153.9s
$S_5 \wr S_{100}$	8.8s	515.5s	2837.0s
$\mathrm{Fi}_{23} \wr S_2$	52.1s	4.1s	70.2s
Fi_{23}	13.4s	0.4s	10.9s
diag $A_5 \wr P$	2.1s	0.054s	0.2s

In the column marked G we mean by S_n the symmetric group on n points, with A_5 the alternating group on 5 points, with Fi_{23} the sporadic simple Fischer 23 group on 31671 points and with diag $A_5 \wr P$ a group which acts on 5000 points with a primitive group (`PrimitiveGroup(1000,7)` in the GAP library of primitive groups) acting on a block system with 1000 blocks with 5 points each and with a block stabilizer isomorphic to A_5 acting simultaneously on all blocks.

The first three groups are large-base groups and clearly the recursive approach shows its merits. The last three groups admit a short base. Our recognition approach is somewhat slower than the traditional stabilizer chain approach because we first have to exclude the possibility of a large-base input before reverting to the small-base methods. In the case when the group order is given a priori (this is the case when we are setting up a data structure for the study of a "known" group), it is known that the verification phase of the stabilizer chain computation can be sped up. We may not realize similar savings with our approach because usually we have no a priori knowledge of the order of each homomorphic image we work with.

3.2 Matrix groups

Computations with matrix groups are more difficult than with permutation groups. One of the main reasons is that there is no analogue of the Schreier–Sims algorithm which runs in polynomial time in the input length. There are two basic methods dealing with matrix groups: the *geometric approach*, summarized in [13], and the *black-box approach* [2]. The primary motivation for the development of our data structure was to provide a framework for computations in the geometric approach.

The geometric approach is based on Aschbacher's classification [1] of subgroups of $\mathrm{GL}(d, q)$. There are nine categories, and the groups G belonging to seven of them preserve some geometric structure and there is a normal subgroup $N \lhd G$ naturally associated with the geometric structure. For example, there may be an invariant subspace W of the

underlying vector space $\mathrm{GF}(q)^d$, and N is the kernel of the action on W; or $\mathrm{GF}(q)^d$ can be decomposed as a direct sum of subspaces such that these subspaces are permuted by G, etc. The goal is to develop procedures which recognize that a group G belongs to these seven Aschbacher categories, and implement corresponding find homomorphism methods using $\varphi : G \to G/N$. Such groups lead to inner nodes in our tree data structure. Some of these homomorphisms map G onto a permutation group, or into such a small matrix group that we can write a permutation representation for the image, in time polynomial in the input length. The other two Aschbacher categories contain almost simple groups modulo scalars: the classical groups (linear, unitary, orthogonal, and symplectic) in their natural representation, and the other representations of almost simple groups, respectively. Groups in these categories are handled by solving the constructive membership problem, and lead to leaves in our data structure.

Currently, we have find homomorphism methods for four of the seven reductive Aschbacher categories and for some of the almost simple groups in arbitrary representations. Work on reduction algorithms for the remaining three reductive Aschbacher categories is under way, and will be finished soon. The major task is to have more algorithms in the almost simple case. Besides our code and subroutines from the *GAP* library, we incorporated programs written by P. Brooksbank, F. Lübeck, S. Howe, M. Law, and A. Niemeyer.

4. ACKNOWLEDGMENTS

This research started when the authors were visiting the School of Mathematics and Statistics at the University of Western Australia. We would like to thank the hospitality of UWA and the support by an ARC Large Grant. The second author is partially supported by the NSA and NSF.

5. REFERENCES

[1] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Invent. Math.*, 76:469–514, 1984.

[2] L. Babai and R. Beals. A polynomial-time theory of black box groups I. In *Groups St. Andrews* 1997 *in Bath, I*, volume 260 of *London Math. Soc. Lecture Note Ser.*, pages 30–64, 1999.

[3] L. Babai, G. Cooperman, L. Finkelstein, E. M. Luks, and Á. Seress. Fast Monte Carlo algorithms for permutation groups. *J. Comp. Syst. Sci.*, 50:296–308, 1995.

[4] L. Babai, G. Cooperman, L. Finkelstein, and Á. Seress. Nearly linear time algorithms for permutation groups with a small base. In *Proc. of Intern. Symp. on Symbolic and Algebraic Computation ISSAC '91*, pages 200–209. ACM Press, 1991.

[5] L. Babai, A. J. Goodman, W. M. Kantor, E. M. Luks, and P. P. Pálfy. Short presentations for finite groups. *J. Algebra*, 194:97–112, 1997.

[6] L. Babai, E. M. Luks, and Á. Seress. Permutation groups in NC. In *Proc. 19th ACM STOC*, pages 409–420. ACM Press, 1987.

[7] L. Babai, E. M. Luks, and Á. Seress. Fast management of permutation groups I. *SIAM J. Computing*, 26:1310–1342, 1997.

[8] R. Beals, C. Leedham-Green, A. Niemeyer, C. Praeger, and Á. Seress. A black-box group algorithm for recognizing finite symmetric and alternating groups, I. *Trans. AMS*, 355:2097–2113, 2003.

[9] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *J. Symbolic Comput.* 24:235–265, 1997.

[10] G. Cooperman and L. Finkelstein. Combinatorial tools for computational group theory. In *Groups and Computation*, volume 11 of *Amer. Math. Soc. DIMACS Series*, pages 53–86. AMS, 1993.

[11] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version* 4.4. Aachen–St Andrews, 2004.

[12] M. Law, A. Niemeyer, C. Praeger, and Á. Seress. A reduction algorithm for large-base primitive permutation groups. *London Math. Soc. J. Computational Math.*, to appear.

[13] C. R. Leedham-Green. The computational matrix group project. In *Groups and Computation III*, volume 8 of *OSU Mathematical Research Institute Publications*, pages 229–247. Walter de Gruyter, 2001.

[14] E. M. Luks. Parallel algorithms for permutation groups and graph isomorphism. In *Proc. 27th IEEE FOCS*, pages 292–302. IEEE Comp. Soc. Press, 1986.

[15] J. J. Rotman. *An Introduction to the Theory of Groups*. Springer, 4th edition, 1995.

[16] Á. Seress. *Permutation Group Algorithms*. Cambridge University Press, Cambridge, 2003.

[17] C. C. Sims. Computational methods in the study of permutation groups. In *Computational Problems in Abstract Algebra*, pages 169–183. Pergamon Press, 1970.

[18] C. C. Sims. Computing the order of a solvable permutation group. *J. Symbolic Comput.* 9:699–705, 1990.