Enumerating large orbits and direct condensation

Frank Lübeck and Max Neunhöffer

Abstract

We describe a new implementation of *direct condensation*, which is a tool in computational representation theory. The crucial point for this is the enumeration of very large orbits for a group acting on some set. We present a variation of the standard orbit enumeration algorithm that reduces the amount of storage needed and behaves well under parallelization. For the special case of matrices acting on a finite vector space an efficient implementation is described. This allows to use condensation methods for considerably larger permutation representations as could be handled before.

1 Introduction

Notation. Let $G = \langle g_1, \ldots, g_r \rangle$ be a group given by r generators. Let M be a set, $\mathrm{S}ym(M)$ the symmetric group on M and $\pi : G \to \mathrm{S}ym(M)$ a homomorphism. We say that G *is acting on* M, and for $m \in M$, $g \in G$ we write $mg := \pi(g)(m)$. Clearly, π is uniquely determined by the images $\pi(g_i)$, $1 \le i \le r$, of its generators. We call the elements of M *points*, and for $m \in M$ the set $mG := \{mg \mid g \in G\}$ is called the *G-orbit of m (under the action π)*. Let $m_0 \in M$ such that its G-orbit m_0G is finite.

Let K be a subgroup of G, also given by a finite number of generators. The main purpose of this paper is the discussion of an algorithm which computes for each K-orbit $mK \subset m_0G$, $m \in m_0G$, the intersection numbers of its translates mKg_i with all such K-orbits. This is explained in section 2 which starts with the description of two basic algorithms for enumerating an orbit.

An interesting interpretation of these numbers in the representation theory of the group G is shortly explained in section 5, which also contains two explicit examples. This application, called *direct condensation*, was our original motivation for this note. But we hope that our general remarks about enumerating large orbits will be useful for other applications as well.

The result in 5.3 is of independent interest because it can be used to finish the determination of the decomposition numbers of the symmetric groups S_n, $21 \leq n \leq 23$, in characteristic 5. These were previously unknown.

The emphasis in algorithm 2.5 is to give it in a way which makes it easy to find variations allowing the practical application to very large cases. For example it may not be possible to store all points of the orbit on a computer because it would need too much memory. Therefore we introduce a concept of *minimal* points in an orbit and only those need to be stored during the algorithm.

In section 3 we make this explicit in the important practical case were G is acting via matrices on a (finite) vector space. We show a way to define minimal elements which allows to get very efficiently from an arbitrary vector in the orbit to a minimal one in the same K-orbit. This is the key point which makes our algorithm for direct condensation much more efficient than a previous implementation described in [2] (see section 5.2 for more details). In section 4 we discuss a parallelization of the algorithm and our implementation. This can be used to treat substantially larger cases as could be handled before.

Acknowledgement. We would like to thank J. Müller for very useful discussions on the topic.

2 The orbit algorithm and variations

In this section we first describe algorithms to enumerate the G-orbit m_0G from the given $m_0 \in M$ and given $\pi(g_i)$, $1 \leq i \leq r$. (Recall that we assumed that m_0G is finite.)

2.1 The basic algorithm

We start with the basic algorithm.

Algorithm 2.1 `Orbit`
Input: $m_0 \in M$, $\pi(g_i)$, $1 \leq i \leq r$

Output: a list L containing the elements of m_0G
Initialize: $L := [m_0]$
for m in L **do**
  for i from 1 to r **do**
    $x := mg_i$
    if x is not contained in L **then**
      append x to the list L
    end if
  end for
end for
return L

The algorithm terminates because of the assumption that the orbit m_0G is finite. It is clear that the resulting list L contains only elements from the G-orbit of m_0. Furthermore L is invariant under the action of the generators g_i and so under their inverses g_i^{-1}. Since each element of G is a finite product of these generators and their inverses L is invariant under the action of G. This shows that L contains exactly the elements of the orbit m_0G.

Counting the number of necessary operations in the algorithm we find:

Proposition 2.2 *The algorithm 2.1 needs r times the length of m_0G operations consisting of an application of a generator of G to a point and a lookup of the resulting point in the list of known points.*

Note that with a naive implementation of the lookup of points in L by sequential comparison the lookup part of an operation described in the proposition would take most of the running time in case of large orbits. But using a standard technique like hash tables, see [5, 6.4], the time for a single lookup becomes (almost) independent of the length of the orbit.

2.2 Large orbits

Our interest here is the practical enumeration of large orbits. Two problems arise: the list of all points in an orbit does not fit into the computer memory and the running time of the algorithm may be longer than we want to wait for the result.

Both problems are addressed by the following variation of algorithm 2.1.

We will now assume that we have a partition of M which is a refinement of the partition into G-orbits. As a typical example think of the partition into orbits under a subgroup of G. Furthermore we assume that for each part a non-empty subset is defined whose points we will call *minimal*. Finally we assume in the following algorithm that we have three functions available:

- `part`, which returns for a given $m \in M$ a list of the points in the part containing m.
- `minimals`, which returns for a given $m \in M$ a list of the minimal points of `part`(m).
- `minimal`, which returns for a given $m \in M$ one minimal point in `part`(m).

To save computer memory for the orbit m_0G the following algorithm will compute only a list of its minimal points.

Algorithm 2.3 `OrbitByPartition`
Input: $m_0 \in M$, $\pi(g_i)$, $1 \leq i \leq r$
Output: a list L containing the minimal elements of m_0G with one element of each part marked as representative
Initialize: $L :=$ `minimals`(m_0) and mark first element as representative
for m' in L which is marked as representative **do**
    $L_1 :=$ `part`(m')
    for m in L_1 **do**
        for i from 1 to r **do**
            $x :=$ `minimal`(mg_i)
            if x is not contained in L **then**
                append the elements of `minimals`(x) to L and
                mark one of the new points as representative
            end if
        end for
    end for
end for
return L

Note that with the output L of this algorithm it is possible to run through all points in m_0G using the function `part` as above. Also, one can check for an arbitrary point $m \in M$ whether it is contained in m_0G by checking whether `minimal`(m) is in L.

The order in which the parts are handled in the outer loop of this algorithm does not matter (except for the ordering of the points in the resulting list L). We will show in section 4 how to use this for parallelizing the algorithm.

2.3 Orbit intersection matrices

Let $K = \langle k_1, \ldots, k_s \rangle$ be a subgroup of G given by s generators and let $m_1, \ldots, m_m$ be representatives of the K-orbits within m_0G.

Definition 2.4 In the setting above we call the matrices $(a_{kl}(g_i))_{1\leq k,l\leq m}$ with

$$a_{kl}(g_i) := |m_kKg_i \cap m_lK|\,,$$

for $1 \leq i \leq r$, the *K-orbit intersection matrices of the g_i on m_0G.*

We are interested in the practical computation of these orbit intersection matrices. In section 5 we will discuss an application of these matrices in the representation theory of the group G.

From now on we will assume that the partition of M described in 2.2 is a refinement of its partition into K-orbits.

In the following algorithm we use 2.3 `OrbitByPartition` in two ways: First, we use it for the whole orbit m_0G, the partition being given by the K-orbits, and supplemented by some bookkeeping for the orbit intersection matrices. And second, we use it as it is, the parts being as before, to compute the K-orbits.

Algorithm 2.5 `OrbitIntersectionMatrices`
Input: m_0, $\pi(g_i)$, $1 \leq i \leq r$, $\pi(k_j)$, $1 \leq j \leq s$
Output:

- a list L containing the minimal elements of m_0G with one element of each part marked as representative
- a map `nr` $: L \to \{1, \ldots, m\}$ with `nr`$(a) = l$ if $a \in m_lK$
- the orbit intersection matrices $A^{(i)} := (a_{kl}(g_i))$ for $1 \leq i \leq r$

Initialize:

- $k := 1$ (loop variable for number of K-orbit)
- $L :=$ `OrbitByPartition`$(m_0, \pi(k_1), \ldots, \pi(k_s))$ (start with minimal elements in first K-orbit m_0K)
- $n := 1$ (number of last found K-orbit)
- `nr`$(a) := 1$ for all $a \in L$
- $A^{(i)} := (0)$ for $1 \leq i \leq r$ (initialize $A^{(i)}$ with 1×1 zero matrices)

while $k \leq n$ **do**

(loop over K-orbits in m_0G, we evaluate only one of its parts at one time)

$L_1 :=$ list of $a \in L$ with `nr`$(a) = k$

for m' in L_1 which is marked as representative of its part **do**
    $L_2 :=$ `part`(m')
    for m in L_2 **do**
        for i from 1 to r **do**
            $x :=$ `minimal`(mg_i)
            if x is not contained in L **then**
                (K-orbit xK is not yet known, we compute it now)
                $n := n + 1$
                append `OrbitByPartition`$(x, \pi(k_1), \dots, \pi(k_s))$ to L
                set `nr`$(a) := n$ for the new points in L
                enlarge all $A^{(j)}$, $1 \leq j \leq r$, by a zero-column and -row
                $A^{(i)}_{kn} := 1$
            else
                (in this case we know the number of the K-orbit of x)
                $l :=$ `nr`(x)
                $A^{(i)}_{kl} := A^{(i)}_{kl} + 1$
            end if
        end for
    end for
end for
end while
return L, `nr` and $A^{(i)}$ for $1 \leq i \leq r$

Note that in this algorithm only the minimal points of m_0G plus the points of one part at a time have to be stored. Typical orbit intersection matrices are dense. That means that during the execution of this algorithm there are two phases. During the first phase mainly new K-orbits are evaluated. After the computation of the first few rows of the orbit intersection matrices the list L is complete. In the second phase the remaining part of the orbit intersection matrices is determined.

Remark 2.6 Assume that we have already run the algorithm 2.5 once and want to know the orbit intersection matrices for additional elements of G. This can be achieved by a small modification of the previous algorithm: Input are the elements of G whose orbit intersection matrices we want to know and the resulting L and `nr` from a previous call of `OrbitIntersectionMatrices`. The only difference is now that in the initialization L and `nr` are set to the given ones. Of course, here the case $x \notin L$ in the inner loop never occurs.

3 The case of matrices acting on vectors

One case for the setup in section 2 is that the set M is a finite dimensional vector (row) space over a finite field and the action of the generators g_i (and the k_j) is described by matrices acting on M by right multiplication.

There is an implementation of a parallelized algorithm for computing orbit intersection matrices in this case by G. Cooperman and M. Tselman, see [2]. An important aspect of their algorithm is also to save memory by not storing all points in the orbit. But if the proportion of stored elements in such an orbit is $1/\alpha$, then one needs in average about α vector-matrix multiplications to find from an arbitrary point in this K-orbit one of the stored points. This essentially leads to a multiplication of the total running time of the algorithm by a factor α.

Another approach was taken by R. Parker and R. Wilson who have a (sequential) program which uses "tadpoles" for saving memory. Since there does not seem to exist any reference for this, here is the idea: One defines a "random-looking" successor function on the set of points and stores only "attracting points" under repeated application of this function. Under certain statistical assumptions one expects to pay for a saving factor $1/\alpha$ in memory usage with a $\log(\alpha)$ time penalty factor. But it seems to be very difficult to predict the behavior of the algorithm in practical cases.

We will now explain a way to realize our functions `minimals`, `minimal` and `part` described in 2.2 efficiently for this case. This allows to reduce the needed memory by a large factor. But the computing time is only increased by a small constant factor compared to the basic orbit algorithm.

We consider a subgroup U of K with the following properties:
(1) U is small enough that we can store all elements of U in our process for computing the orbit intersection matrices
(2) there is a U-invariant subspace V of M such that all U-orbits of the quotient space M/V can easily be computed
(3) the average length of the U-orbits on M/V is "close to" $|U|$

In practical examples it turned out that it is not difficult to find such U and V. The space M viewed as a G-module is typically irreducible. Small subgroups of K as candidates for U can be found by considering some subgroups generated by random elements. Now M considered as U-module usually has a composition series consisting of many small dimensional modules. This can be found using the MeatAxe, see [9], and so we find candidates for V.

Assume that we have found U and V as above. Let $\texttt{pr} : M \to M/V$ be the projection map. Note that the action of U on M and the induced action on M/V commute with pr. We enumerate M/V and call $m \in M$ *minimal*, if $\texttt{pr}(m)$ is minimal in its U-orbit with respect to this enumeration. (pr is particularly easy to implement when the basis of M is chosen to contain a basis of V.)

In a precomputation (short, because of (2)) we compute by a variation of the basic orbit algorithm 2.1 for each point of M/V either an element of U mapping it onto the minimal element in its U-orbit or, if the point is already minimal, the elements of U stabilizing this point.

Now we implement $\texttt{part}(m)$ by computing all mu, $u \in U$, and removing multiple points. Because of (3) this takes not much more than one vector-matrix multiplication per element in $\texttt{part}(m)$.

For $\texttt{minimal}(m)$ we use the precomputation, for $\texttt{pr}(m)$ we have stored a $u \in U$ such that mu is minimal. All $\texttt{minimals}(m)$ are computed by applying all u' from the stabilizer of $\texttt{pr}(mu)$ to mu and removing multiples. (Because of (3) this stabilizer is often trivial.)

Using these considerations we can count the basic operations which are needed in 2.5.

Proposition 3.1 *We assume that in the situation above the computation of* $\texttt{minimals}(m)$ *takes in average less than* 2 *vector-matrix multiplications and that the computation of* $\texttt{part}(m)$ *takes in average less than* 2 *vector-matrix multiplications per point in the part.*

(a) Then the algorithm 2.3 `OrbitByPartition` *needs* $2+2r$ *vector-matrix multiplications and* r *list lookups per point in the considered orbit (neglecting the computation of all minimal points once for each part).*

(b) Then the algorithm 2.5 `OrbitIntersectionMatrices` *needs* $(2+2s)+(2+2r)$ *vector-matrix multiplications and* $s+r$ *list lookups per point in the considered orbit (here we neglect the calls to* `minimals` *once for each part and the bookkeeping effort for the orbit intersection matrices).*

We remark that a similar idea also works in the case of G acting on the subspaces of M, instead of the vectors.

In certain cases one can think of further improvements by choosing subgroups U with additional nice properties. For example, if M is a semisimple U-module then one can take a basis of M such that elements of U have a (very sparse) block diagonal form. (In general one can reach a block triangular form.)

4 Parallelization

We do not see an improvement of algorithm 2.5 `OrbitIntersectionMatrices` which reduces the computation time considerably. But we can reduce the waiting time for the result by distributing the computations in parallel among several computer processors. We are mainly thinking of using networks of workstations. In this section we describe our approach to a parallelization of `OrbitIntersectionMatrices`.

4.1 Parallel version of `OrbitIntersectionMatrices`

Looking at the algorithm we see that it is essential to have a central place where the list L and the map `nr` are managed (in form of a hash table) to avoid many computations of the same K-orbits by several processes and also to guarantee a unique numbering of the K-orbits found.

We divide the whole work into pieces by giving single runs through a K-orbit as jobs to single processors. In such a job - corresponding to a run through the body of the outer loop in algorithm 2.5 - one row of each of the orbit intersection matrices is computed.

We have written a small library which allows communication of processes running on computers connected via a network (using UNIX domain sockets, which are available on many computer operating systems). The communication is of the type that one process sends to another a number indicating a type of a request plus some data. The other process may do some computation and then sends back an answer in form of a block of data. Using this we have implemented three different programs which work together.

First there is one process called the ***jobserver***: It can be asked for a job to do (a number k in algorithm 2.5), or for a number for a newly found K-orbit, and it collects the computed rows of the orbit intersection matrices and stores them into files.

Then there is one process (or several, see below) called the ***hashserver***: This one manages the hash table for the list L. It can be asked to send for a given list of points the corresponding numbers of their K-orbits or the information which points are lying in a not yet known K-orbit. Also this process can be asked to store the information about a new K-orbit in its hash table (it writes it to a file, too), and also to send a list of representatives for the parts which are contained in the K-orbit with a given number.

Finally there can be many processes called ***dcclient***: They ask the ***jobserver*** for a job, get the representatives for the K-orbit they have to handle from the ***hashserver***, then run through the body of the outer loop of algorithm 2.5, send the computed rows of the orbit intersection matrices to the ***jobserver***, and start from the beginning. When such a process has to check whether a point is contained in L and wants to know the number of its K-orbit then it sends the point to the ***hashserver*** to get the answer. When a new K-orbit is computed it is sent to the ***hashserver*** (which ignores it in the rare case that this K-orbit was in the meanwhile already computed by another process). Actually a ***dcclient*** does not send single points as requests to the ***hashserver*** but always computes `minimal`(mg_i) for all m in a fixed part and puts a collected request to the ***hashserver*** into a queue. Before computing the next such request it checks for available answers from the ***hashserver***. This way the client process does not have to be idle in case of a temporarily overloaded network.

As a variant we also allow ***multiple hashservers***: Here we use a function which computes for a given point the number of a ***hashserver*** which is responsible to store this point and to answer requests about it. This makes the preparation of ***hashserver*** requests by a ***dcclient*** slightly more complicated but it can be very useful in certain situations: For example if the data for the requests are computed so fast that the network bandwidth is too small and if we have a switched network (which allows several parallel connections with full bandwidth) then ***multiple hashservers*** can increase the overall available network bandwidth for the requests. And this is similar when a ***hashserver*** cannot handle all the requests fast enough. Another point is that the ***hashserver*** is usually the process which needs most of the memory. For efficiency it is desirable that a ***hashserver*** can keep the list L in the physical memory of the computer. Using ***multiple hashservers*** we can use the physical memory of several computers for this purpose.

Concerning the memory needed by these processes the hash servers need to hold all minimal points of m_0G and a client process needs to store at most all minimal elements of a K-orbit and the points in one part of the partition. The orbit intersection matrices can also become very big (there may be up to 10000 K-orbits, say). But it is never necessary to store more than one row in a client. Once a row is computed it can be stored in a file and is not needed any more. (We only have to append some zeros when we use them, because some new K-orbits can be found after finishing a row.)

The computation time for the algorithm scales almost linearly with the number of clients as long as the bandwidth of the communication between the clients and the hash servers or the computing power of the hash servers do not reach their limit. And the amounts of data which have to be transfered can be estimated very well from 3.1. If network bandwidth becomes a problem one can at least speed up linearly the second phase of the algorithm described after 2.5: We interrupt the computation after finding all K-orbits and start it again in the form of 2.6 with several hash servers who all use the same already computed data.

4.2 Comments on the implementation

Our implementation of the parallel version of `OrbitIntersectionMatrices` is written as far as possible in a generic way (the programming language is C), where we assume almost nothing about how the points of M, elements of G and the action are given. To get a program for a special case one has to write a file containing functions for initializing the clients, operation of group elements on points, the functions `part`, `minimal` and `minimals`, and hash functions for the points. This can then be linked easily with the main part of the program.

The part for the client-server communication is a separate small package which can be used for other programs as well. It supports simple blocking requests, i.e., where a process waits for an answer, as well as queues of non-blocking requests.

An advantage of our communication approach seems to be robustness: The crash of any single process involved in a computation does not waste the computing time spent so far. Client processes can be terminated and new ones started up at any time. Of course the whole computation crashes when one of the server processes is terminated for some reason. But we are saving the results which are already obtained into files and this makes it possible to restart the computation almost at the point where it was stopped. This feature is very important for the use of such programs on networks where any single machine can be down at any time for various reasons. The lack of this feature was also the reason that we did not use a (in certain aspects much more sophisticated) communication protocol like MPI, see [8].

The initial revision of our package contains two versions of the programs. One with permutations as group elements for doing the computation described in 5.3 and another more general one for matrices acting on vectors over a finite field. In the latter we use some basic functions from M. Ringe's

MeatAxe package, see [9]. Since we want to use this program for very large examples we have put some effort in optimizing the vector-matrix arithmetic, e.g., by precomputing certain linear combinations of rows of the operating matrices (R. Parker calls this "greasing") and by using partial row operations for sparse rows.

Our software is freely available under GPL [4], see [6].

5 Direct condensation

Let A be a finite dimensional algebra over a field F and let $e = e \cdot e \in A$ an idempotent. The idea of *condensation* is to get information on A-modules $\mathcal{M}$ by studying the eAe-modules $\mathcal{M}e$. In particular this is an important tool in computational representation theory. The latter modules can be of much smaller dimension but still encode interesting information on the structure of $\mathcal{M}$, since the map $\mathcal{M} \mapsto \mathcal{M}e$ is an exact functor from the category of A-modules to the category of eAe-modules.

For more details we refer to [1] and the references given there. The first reference describing the use of this method in modular representation theory is J. Thackray's thesis [10].

5.1 Interpretation of the `OrbitIntersectionMatrices`

We want to consider the special case when $A = FG$ is the group algebra of a finite group G over the field F, e is the idempotent $1/|K| \cdot \sum_{k \in K} k \in FG$ corresponding to the subgroup K of G whose order is not a multiple of the characteristic of F, and $\mathcal{M}$ is a permutation module of FG. (If e is of this form then K is called the *condensation subgroup*).

Now we assume that G and M are finite. A permutation representation $G \to \mathrm{Sym}(M)$ of G describes a permutation module $\mathcal{M}$ of FG. A basis for this module is parameterized by the elements of M. Let $x = \sum_{m \in M} a_m m \in \mathcal{M}$ and O be a K-orbit of M. Then for all $m \in O$ the coefficient of m in xe is $1/|O| \cdot \sum_{m' \in O} a_{m'}$. This shows that the orbit sums $\chi_O := \sum_{m \in O} m$ for all K-orbits in M are a basis of $\mathcal{M}e$. Furthermore we see how for $g \in G$ the element ege is acting on this basis: for another K-orbit O' the coefficient of $\chi_{O'}$ in $\chi_O ege$ is $1/|O'| \cdot a_{O,O'}$ with $a_{O,O'} := |\{m \in O \mid mg \in O'\}|$.

Clearly $\mathcal{M}$ is a direct sum of the permutation modules on the G-orbits in M. Our algorithm 2.5 `OrbitIntersectionMatrices` computes exactly the numbers $a_{O,O'}$ for all K-orbits in a single G-orbit. (Note that the sum of entries in a fixed row or column of an orbit intersection matrix gives the length of the corresponding K-orbit.) The method was called *direct condensation* by R. Parker and R. Wilson because one only needs to know for a given point $m \in M$ and $g \in G$ its image mg but one does not need to write down in full detail the explicit permutation induced by g on M.

5.2 An application with $G = \mathrm{Th}$

As first example for our program we have checked the computations in [1]. There G is the sporadic simple Thompson group acting linearly on a vector space M of dimension 248 over the field with 2 elements. The considered G-orbit has about 10^9 elements. The cited paper contains enough details that we could redo the computations starting with the matrices for this representation given in R. Wilson's WWW-Atlas of group representations, see [11].

We used the approach described in section 3. As subgroup U for the partition of the orbit we constructed a group of order 336 which has an invariant subspace in M of codimension 20. It turned out that about 1 out of 257 points in the considered G-orbit is minimal. The minimal vectors can be stored in 125 Megabytes of memory using one bit per field element.

After measuring the time needed for a single vector-matrix multiplication we estimated the total running time of the condensation using Proposition 3.1. We found that this estimate was very close to the actual running time. The computations were done on 18 machines (Pentium II, 450MHz processors) of a cluster at the university of St.Andrews[1] which are connected by a "switched fast ethernet network". We used one ***hashserver*** which had to handle about 65 Gigabytes of lookup requests. The computation needed less than 4 hours. (To compare with [1, 3.3]: There 8 machines computed for one month - a single vector-matrix multiplication took about the same time as in our case - and 610 Megabytes of vectors had to be stored.)

We have also done some larger computations for other sporadic simple groups. The results can hopefully contribute to the determination of the modular character tables of these groups. Details will be given elsewhere.

[1]This hardware was provided by EPSRC grant GR/M32351.

5.3 An application with $G = S_{21}$ a symmetric group

As another application we condensed the permutation module of a Young subgroup of type $(8, 8, 4, 1)$ in the symmetric group $G = S_{21}$. The motivation was a question by G. James and A. Mathas who could determine the decomposition matrix for the irreducible representations of G in characteristic 5 up to a single entry. The question was whether in the Specht module of G labelled by the partition $(8, 8, 4, 1)$ reduced modulo 5 the irreducible module labelled by $(12, 9)$ occurs once or twice.

J. Müller has found a subgroup K of G with the property that the permutation module of type $(8, 8, 4, 1)$ condensed with K as condensation subgroup has either 761 or 762 constituents in a composition series depending on the two possible cases. Such a consideration can be made using only the two possible tables of Brauer characters for G and the character table of K. (See again [1] for a more detailed explanation.) The group K is a transitive subgroup of order $47,029,248$ which has the number 147 in the database of transitive groups contained in GAP [3].

In this case our M consists of 21-tuples of numbers on which G acts by permutation. The permutation module we want to condense is described by the orbit of $[0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3]$. It has dimension $1,309,458,150$. As generators of G we took two random elements and K was also given by two generating permutations. As partition of the orbit for algorithm 2.5 we took again orbits under a subgroup of K. There is a nice subgroup which is a direct product of 7 symmetric groups S_3, each factor permuting the entries of 3 consecutive positions. We define for each part a unique minimal point, namely that one whose entries are sorted in positions $\{1, 2, 3\}$, $\{4, 5, 6\}$ and so on. There are only about $660,000$ minimal points in the orbit. So, the operation of group elements on the points and the functions `part`, `minimal` and `minimals` are implemented easily and efficiently for this case.

In this example the operation of group elements on the points can be computed so fast that already in a parallel computation with a few clients the network bandwidth can be reached with the requests for the ***hashserver***. Here our approach with ***multiple hashservers*** improves the situation considerably. On 20 machines of the network already mentioned in 5.2 using 20 client and 20 hash server processes the orbit intersection matrices for the two generators of G could be computed in 38 minutes and during this time about 96 Gigabytes of data were sent over the network.

The condensed module has dimension 4197. A composition series of this module for the algebra generated by the two computed elements of form ege can be found with the MeatAxe [9] within a few hours of computation time. We found 761 constituents and this rules out the possibility for the decomposition number which would imply 762 constituents. (Note that it is not clear whether our two elements ege generate the whole algebra $eFGe$, but taking further generators into account cannot increase the number of constituents.)

Proposition 5.1 *Let $G = S_{21}$ and F be a field of characteristic 5. The multiplicity of the simple FG-module labeled by the partition $(12, 9)$ in the FG-Specht module labeled by $(8, 8, 4, 1)$ is one.*

This result together with work of A. Mathas and G. James, in particular the software package Specht [7], determine the decomposition numbers in characteristic 5 for all symmetric groups S_n with $n \leq 23$.

References

[1] G. Cooperman, G. Hiss, K. Lux, and J. Müller. The Brauer tree of the principal 19-block of the sporadic simple Thompson group. *Experimental Mathematics*, 6(4):293–300, 1997.

[2] G. Cooperman and M. Tselman. New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. In Y. N. Lakshman, editor, *Proceedings of ISSAC 96*, pages 155–160, New York, 1996. ACM Press.

[3] The GAP Group, Aachen, St Andrews. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 1999. (see `http://www-gap.dcs.st-and.ac.uk/~gap`).

[4] GNU General Public License. see `http://www.gnu.org/copyleft/gpl.html`.

[5] D. E. Knuth. *The Art of Computer Programming*, volume 3, Sorting and Searching. Addison Wesley, second edition edition, 1997.

[6] F. Lübeck and M. Neunhöffer. *DC – a package for Direct Condensation programs*. Lehrstuhl D für Mathematik, RWTH Aachen, 2000. (see `http://www.math.rwth-aachen.de/~DC`).

[7] A. Mathas. Specht - *Decomposition matrices for the Hecke algebras of type A (manual for version 2.4)*. University of Sydney, 1997. (see `http://www.maths.usyd.edu.au:8000/u/mathas/specht/`).

[8] MPI, collected information. see `http://www-unix.mcs.anl.gov/mpi/`.

[9] M. Ringe. *The C-MeatAxe, a manual*. Lehrstuhl D für Mathematik, RWTH Aachen, 1998. (see `http://www.math.rwth-aachen.de/~MTX/`).

[10] J. G. Thackray. *Modular representations of finite groups*. Ph.d. thesis, Cambridge University, 1981.

[11] R. Wilson. WWW-Atlas of group representations. (see `http://www.mat.bham.ac.uc/atlas/`).

Authors: Frank Lübeck, Max Neunhöffer
Mail address: Lehrstuhl D für Mathematik, RWTH Aachen, Templergraben 64, 52062 Aachen, Germany

e-mail: Frank.Luebeck@Math.RWTH-Aachen.De
Max.Neunhoeffer@Math.RWTH-Aachen.De

WWW: `http://www.math.rwth-aachen.de/~Frank.Luebeck`
`http://www.math.rwth-aachen.de/~Max.Neunhoeffer`