March, 1998
This paper appeared in
J. Comb. Maths and Comb. Computing 31 (1999) 193-206.

A Modification of the Schreier-Sims Algorithm Utilising the Transitivity of the Stabiliser Subgroups

William Kocay*
Computer Science Department
University of Manitoba
Winnipeg, Manitoba, CANADA, R3T 2N2
e-mail: bkocay@cs.umanitoba.ca

Abstract

A modification of the Schreier-Sims algorithm is described which builds a permutation group utilising the transitivity of the stabiliser subgroups. Alternating and symmetric groups are recognized by their transitivity, resulting in a greatly improved time to build symmetric and alternating groups. The algorithm has applications to graph isomorphism and other combinatorial isomorphism algorithms and to permutation group algorithms.

1. Introduction

When constructing a permutation group G using the Schreier-Sims algorithm [2,3], it would often be convenient to know the transitivity of G. Once a group has been built as a tower of stabiliser groups, it is fairly straightforward to run through its tower of subgroups and calculate the transitivity by computing all the orbits of the stabilisers. However, there is an advantage to knowing the transitivity at all times as the group is being built. The alternating and symmetric groups are the most time-consuming to build. Yet they are easily recognized by their transitivity. If we knew the transitivity, we could bypass much of the computation normally required to build the group. In this article we show one way to accomplish this.

The data structures and algorithms we use to implement the Schreier-Sims algorithm is that of [4]. A group is stored as a tower of stabilisers. Each group in the tower is represented by a record containing: generators for the stabiliser; the point u whose stabiliser is next in the tower; the orbit of u; an array of coset representatives, one for each point in the orbit.

* This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

GroupPtr = ^Group
Group = record
    Generators: linked list of generators
    u: integer, the point whose stabiliser is next in the tower
    OrbitQueue: integer array, the orbit of u
    nPts: integer, the number of points in orbit of u
    CosetRep: array of pointers to coset reps of G_u
    InverseRep: array of pointers to inverses of coset reps
    G_u: GroupPtr, pointer to the stabiliser of u
end

We assume a procedure *GroupElement*(g, G), which accepts a permutation g and a group G, and returns *true* if $g \in G$ and *false* if $g \notin G$. When a group is being built, generators g of the stabiliser subgroup G_u are discovered. There will be a statement that checks if g is a known element of G_u:

    if not *GroupElement*(g, G_u) then *AddGen*(g, G_u)

AddGen(g, G_u) is a recursive procedure that updates the data structures of G_u to reflect the new generator g. This usually means that one or more stabiliser subgroups in the tower will also be updated recursively. We would like to store an integer *Transitivity* in the record of each group. After the procedure *AddGen*(g, G_u) returns, the transitivity of G_u would be known. We would then check if G_u is now known to be a symmetric or alternating group, by testing its value *Transitivity*. If it is, then a quick calculation will determine if G is also a symmetric or alternating group. If it is, this will obviate the need to continue the orbit-building loop in which generators of G are multiplied and possible generators for G_u are constructed. The result will be a much faster algorithm for building symmetric and alternating groups, plus the advantage of knowing the transitivity of every group in the tower. Every time we build a group, we automatically obtain its transitivity, without having to do a separate calculation for it.

2. Algorithm

We assume that a permutation group G acting on a set $\{1, 2, \ldots, n\}$ of n points is being constructed from a list of generating permutations. Each permutation is stored as an integer array of length n. G is to be stored as a tower of stabiliser subgroups. Each stabiliser in the tower also acts on the n points. We will say that a group is *transitive* if all the points it moves fall into one orbit. The remaining orbits consist of fixed points. For each group G we choose a point u moved by G, and construct the stabiliser subgroup G_u.

We define the *transitivity* of G recursively. The identity group has transitivity 0. The transitivity of G is defined to be k, where $k \geq 1$, if G is transitive, and its stabiliser G_u has transitivity $k-1$ and moves the same points as G (except for u.).

We modify the storage of group G to include its transitivity. The i^{th} point in the orbit of u is u_i =*OrbitQueue*$[i]$. *CosetRep* is an array of pointers to permutations. *CosetRep*$[u_i]$ points to a permutation mapping u to u_i. It is the representative stored for the coset of G_u mapping u to u_i. *InverseRep*$[u_i]$ is the inverse permutation. If the inverse is needed, it is stored here. Otherwise its value is *nil*.

GroupPtr = ^Group
Group = record
    Generators: linked list of generators
    u: integer, the point whose stabiliser is next in the tower
    OrbitQueue: integer array, the orbit of u
    nPts: integer, the number of points in orbit of u
    Transitivity: integer
    CosetRep: array of pointers to coset reps of G_u
    InverseRep: array of pointers to inverses of coset reps
    G_u: GroupPtr, pointer to the stabiliser of u
end

Group G is built by a procedure *AddGen*(g, G), which applies the new generator g to each point in the orbit of u. Either the orbit of u is extended by g, or else one or more new generators of G_u are discovered. The second possibility causes a recursive call to *AddGen*. At first I attempted a scheme based on using the stored transitivity of G_u and only the points moved by the new generator g. This does not work. It results in unavoidable cases where *all* the generators must be applied to build the orbits, possibly at every level in the recursion. This extra computation would defeat the purpose of using the stored transitivity to improve the computation speed.

It seems like the only way it can be done is to store all the orbits at every level in the recursion, and maintain a count of them. When a new generator g is added to G, orbits will merge. We need to be able to detect which orbit a point is in, and to merge distinct orbits. Therefore we use the *merge-find* data structure (also called *set-union*) to represent the orbits. See [1,4,6]. Note that we *do not need to maintain a list of the points in each orbit*. We then need apply *only the new generator* to the existing orbits to update them.

At each level in the recursion, we divide the orbits into three sets – the orbit containing u, which is stored as an array *OrbitQueue*, the fixed points or singleton orbits, and the remaining orbits. We call these remaining orbits

"outside" orbits.

In order to keep a count of the number of outside orbits, the value of *Transitivity* stored for a group G is

$= 0$, if G is the identity group
$=$ transitivity of G, if G is transitve
$= -$(the number of outside orbits), if G is intransitive

In this way, we know that G is transitive if *Transitivity*> 0, and we know the number of outside orbits if *Transitivity*< 0.

In order to store all the outside orbits using an MF-type data structure without adding another array to the group record, we can use the value of *CosetRep*$[v]$ for points v not in the orbit of u. Normally the value of *CosetRep*$[v]$ would be *nil* if v is not in the orbit of u. Instead of *nil* we will now store a value *CosetRep*$[v] = w$, where w is the next point closer to the representative of the orbit of v, if v is in an outside orbit. v is the representative of its orbit if *CosetRep*$[v] = v$. See [4,6] for a description of the merge-find data structure. We use CosetRep$[v] = 0$ to indicate that v is a singleton orbit. In order to distinguish a value *CosetRep*$[v] = w$ from an actual pointer to a permutation, we rely on addresses (pointers) inside a computer being integers not in the range 1 to n. Since this points to a low-memory address, this would not be considered a valid pointer in most computers. If this is not the case for a computer on which this algorithm is implemented, it will be necessary either to use an additional array, or else to modify this method in some other way.

Suppose now that a new permutation g is to be added to group G. The main operation of the Schreier-Sims algorithm is to apply g to every point in the orbit of u. Either the orbit of u is extended, or else a stabiliser subgroup is extended. Then for *all* new points v added to the orbit of u, *every* generator of G is applied to v. Again the orbit may be extended, or a stabiliser subgroup may be extended.

AddGen(g, G)
1. for each $u_i \in$ *OrbitQueue*, find $v := g[u_i]$
2. if $v \notin$*OrbitQueue*, add v to *OrbitQueue*, construct *CosetRep*$[v]$
3. else $v \in$*OrbitQueue*, construct a generator g' for the stabiliser G_u
   if not *GroupElement*(g', G_u) then *AddGen*(g', G_u)
4. for all new points $v \in$ *OrbitQueue*, apply *all* generators of G to v
   as in items 1 to 3.

After *AddGen*(g, G) has been executed, the group G is up to date with respect to the new generator g. We now want to modify this algorithm to simultaneously compute the transitivity of G, and to detect symmetric and alternating groups. As before we still must apply the new generator g

to every point u_i in the orbit of u, and we must apply all generators of G to every new point v added to the orbit. As these computations are proceeding, we must also:

1. keep track of the number of outside orbits
2. after the recursive call to $AddGen(g', G_u)$, we must check if the stabiliser G_u is now known to be alternating or symmetric. If it is, G might also be alternating or symmetric. We check for this, and break out of the orbit building loop if it is indeed the case.

The modified algorithm now looks like this.

AddGen(g, G)
1. for each $u_i \in$ *OrbitQueue*, find $v := g[u_i]$
2. if *CosetRep*$[v] \geq 0$ and *CosetRep*$[v] \leq n$ then $v \notin$*OrbitQueue*
   a. v is either previously a singleton orbit, or in an outside orbit
   b. if *CosetRep*$[v] \neq 0$, v is in an outside orbit which now becomes merged with *Orbit*
      - increment *Transitivity* since there is one fewer outside orbit
      - we must ensure that when the other pts of this outside orbit are added to *OrbitQueue*, that we do not again adjust *Transitivity*
   c. add v to *OrbitQueue* and construct *CosetRep*$[v]$
3. otherwise, $v \in$ *OrbitQueue*, construct a generator g' for the stabiliser G_u
   a. if g' is not currently in G_u, then $AddGen(g', G_u)$
   b. if G_u has *Transitivity*$\geq$ min(2, nPts$-$2) then G_u is $(k-2)$-transitive or more on k points $\Rightarrow$ G_u is either symmetric or alternating.
      - check the *OrbitQueue* of G. If G acts on $k+1$ points, then G is also symmetric or alternating. Decide which and go to 5.
4. for all new points $v \in$ *OrbitQueue*, apply *all* generators of G to v as in items 1 to 3.
5. *CheckTransitivity*(g, G)

There are a number of subtle aspects to this algorithm.

Statement 2.b. of AddGen

We have a point v with *CosetRep*$[v] \neq 0$. v is in an outside orbit which now becomes merged with *OrbitQueue*. We increment the *Transitivity*, since there is now one fewer outside orbit. The other points in the orbit of v will also be encountered during the course of the algorithm. We don't want to increment *Transitivity* again for other points in this orbit. Therefore after merging the orbits we find the reperesentative w of the orbit of v and set *CosetRep*$[w] := 0$. The algorithm will henceforth think that w is a singleton orbit. Since all points in an orbit can be identified via their

representative w, it will not increment *Transitivity* again for this orbit. Somewhere during the course of the orbit-building loop, an actual coset representative will be constructed for w, and for the other new points in the orbit, so that *CosetRep*[w] will be reassigned. So the value of 0 stored is just a temporary value that will correct itself. The procedure that finds the orbit representative must be aware of this usage, and return 0 if *CosetRep*[w] is found to exist. The code that updates the transitivity looks like this.

```
if CosetRep[v] ≥ 0 and CosetRep[v] ≤ n then begin
    { v is either previously a singleton orbit, or in an outside orbit }
    if CosetRep[v] ≠ 0 begin
      w := FindOrbitRep(v)
      if w ≠ 0 then begin
        increment Transitivity
        CosetRep[w] := 0 { the orbit rep of v will now be seen as 0 }
      end
    end
end
```

Statement 3.b. of AddGen

The algorithm has returned from a recursive call to *AddGen*(g', G_u). The stabiliser group G_u is up to date, and so we know if it is alternating or symmetric. This will be the case if it is $(k-2)$-transitive or more, where its *OrbitQueue* contains k points. We require $k-2 \geq 2$ so that it makes sense to speak of G_u being alternating or symmetric.

We have to be careful with the small values of k, since it is the small values that start the induction and make the recursion work properly. Notice that G_u will have been originally initiallized to an identity group. If its first generator happens to be a transposition, it will be seen as 2-transitive on 2 points, a symmetric group. If its first generator happens to be a 3-cycle, it will be seen as 1-transitive on 3 points, an alternating group. This is how we want the algorithm to work.

Assume that after *AddGen*(g',G_u) returns, the group G_u is $(k-2)$-transitive or more on k points. We need to determine whether G is alternating or symmetric or neither. The *OrbitQueue* of G contains nPts points so far, but at this point we don't know the full orbit of u in the group G. Remember, the algorithm is in the middle of the orbit building loop. If nPts $> k+1$, then the orbit of u is too big, and G cannot be symmetric or alternating. Therefore we first check that nPts $\leq k+1$. If this is so, G may be alternating or symmetric, but we can't decide unless we know the entire orbit of u. We must find it at this point. The code looks like this.

```
if G_u has Transitivity ≥ min(2, nPts−2) then begin
    { G_u is (k − 2)-transitive or more on k points }
    if G has nPts ≤ k + 1 then begin
      { G may be alternating or symmetric, we need the full orbit }
      if SymmetricOrAlternating(G, k) then begin
        { G is either symmetric or alternating, decide which }
        if Transitivity+2 = nPts then CheckSymmetric(G)
        go to statement 5
      end
    end
end
```

There are two additional procedure calls here, *SymmetricOrAlternating*(G, k) and *CheckSymmetric*(G). The first one builds the rest of the orbit of u in G, and compares its size against the orbit size k from G_u. If the orbit contains exactly $k+1$ points, that is, exactly one more point than G_u, then since G_u is known to be either alternating or symmetric, and $k \geq 2$, we know that G is also alternating or symmetric. In this case it returns a *true* value. Notice that *SymmetricOrAlternating*(G, k) does not construct coset representatives for the points it adds to the orbit. It merely picks up where *AddGen* left off, and builds the orbit. This is very quick. (Notice that it needs to know at what point in *AddGen* it was called from, so that it can pick up at that point. There is a difference depending on whether it is called from statement 3a or 4. The program must be aware of this.) If it decides that G is in fact alternating or symmetric, it then goes back and constructs coset representatives. It does not construct generators for the stabiliser G_u, since G_u is already known to be alternating or symmetric. If it decides that G is not alternating or symmetric, then it must allow the orbit-building loop of *AddGen* to continue, since *AddGen* may discover further generators for G_u. In this case it resets the values that had been changed. This happens only rarely.

SymmetricOrAlternating(G, k)

1. *AddGen* was last adding $g[u_i]$ to *OrbitQueue*
   G_u is at least $(k-2)$-transitive on k points
   beginning with $j := i + 1$ for each $u_j \in$ *OrbitQueue*, find $v := g[u_j]$
2. if *CosetRep*$[v] \geq 0$ and *CosetRep*$[v] \leq n$ then $v \notin$ *OrbitQueue*
   a. if *OrbitQueue* already contains $k + 1$ points, then go to 5
   b. v is either previously a singleton orbit, or in an outside orbit
   c. if *CosetRep*$[v] \neq 0$, v is in an outside orbit which now becomes merged with *OrbitQueue*
      - increment *Transitivity* since there is one fewer outside orbit
      - we must ensure that when the other pts of this outside orbit

are added to *OrbitQueue*, that we do not again adjust *Transitivity*

c. add v to *OrbitQueue*, do not construct *CosetRep*$[v]$, but save the value u_j so that we can find it again easily later. (Use *InverseRep*$[v] = u_j$)

3. for all new points $v \in$ *OrbitQueue*, apply *all* generators of G to v as in items 1 and 2.
4. the full orbit is now known. We must still check that the orbit of G_u is contained in the orbit of G.
   a. if it is not, then go to 5
   b. otherwise G is either symmetric or alternating
   c. for all points v added to the orbit, construct *CosetRep*$[v]$, using the generator g and the point u_j that was saved in item 2c.
   d. return *true*
5. G is not symmetric or alternating
   a. reset *CosetRep*$[v]$ to 0, for all new points added to *OrbitQueue*
   b. return *false*

Notice that *SymmetricOrAlternating* changes the value of *Transitivity*, which counts the number of outside orbits, but does not reset it if it must return a *false* answer. Once it has merged any outside orbits, it is not necessary to undo this, only to have *AddGen* redo it. Also, *SymmetricOrAlternating* is only called if G_u is found to be symmetric or alternating. If so, there is a very good chance that G will also be found symmetric or alternating, so that it will likely only be called at most once for each group in the tower.

SymmetricOrAlternating also must check if the orbit of G_u is contained in the orbit of G. This is done by testing if the point u saved in the group G_u is in the orbit moved by G. It requires testing one point only, the orbit representative.

The other procedure called by *AddGen* is *CheckSymmetric*(G). It is called after *SymmetricOrAlternating*(G, k) returns *true*. If G_u was discovered to be symmetric, we know that G is also symmetric. Similarly, if G_u was discovered to be alternating, we expect that G is also alternating. However this is not always the case. I discovered the following set of generators for the symmetric group S_6.

$a = (1, 2, 3, 4, 5, 6)$
$b = (1, 3, 5)$
$c = (1, 4)$

If we build a group G by adding first generator a, then b, we get a group of order 6, and then 18. We now call *AddGen*(c, G). *AddGen*

discovers a new generator for the stabiliser G_u. After a recursive call to *AddGen* to extend G_u, it is discovered that G_u is the alternating group A_5. *SymmetricOrAlternating* is then called, and returns *true*, so that G is either symmetric or alternating. If we then assume that G must be alternating, we would be in error. Once it is discovered that G is symmetric or alternating, the orbit building loop of *AddGen* is stopped. In this case, it reaches a point where G_u is A_5. All generators found so far at all levels in the recursion have been even. If it had continued to construct generators for the stabiliser, it would eventually have found an odd one, and extended the stabiliser to S_5. But the whole purpose of *SymmetricOrAlternating* is to avoid all this work precisely for symmetric and alternating groups. Therefore, when *SymmetricOrAlternating* reports alternating, we must check that it really is alternating. One way to do this is by checking whether G has an odd generator. It means extra work for alternating groups, at every level of recursion, since they have no odd generators. However, for groups which are not alternating or symmetric, we never have to check the sign of any generator. For symmetric groups, we stop as soon as we come to the first odd generator.

CheckSymmetric(G)

1. G is either symmetric or alternating, check for an odd generator:
   for each generator g, check if g is odd. If so, go to 3
2. at this point, all generators are even, return
3. an odd generator was encountered – group G is symmetric. All stabilizers in the tower are currently marked as alternating. We must change all stabilisers to symmetric. The group currently at the bottom of the tower is a 3-cycle, say (x, y, z). When point x, say, is fixed, so are y and z. In order to extend this to a symmetric group, we must add the transposition (y, z) as a generator at every level in the tower, and construct a new *CosetRep* for y and z in the group at the bottom of the tower. A new stabiliser is created at the bottom of the tower, generated by the transposition (y, z).

Statement 5 of AddGen.

Statement 5 contains a call to *CheckTransitivity*(g, G), which takes place after G has been made up to date with respect to the new generator g. The value stored in G for *Transitivity* is either the degree of transitivity of G, if G is a transitive group, or else $-$(number of outside orbits), if G is non-transitive. This number must be adjusted if G is not symmetric or alternating. The orbit building loop of *AddGen* has applied g to every $u_i \in$ *OrbitQueue*. We must also apply g to the points not in *OrbitQueue*. Some outside orbits may merge, and others may be created if points previously in singleton orbits are moved by g.

CheckTransitivity(g, G)

1. deltaOrbs := 0 { the change in the number of outside orbits }
2. for each $u \in \{1, 2, \ldots, n\}$ such that $g[u] \neq u$, find $v = g[u]$
3. if *CosetRep*$[u] \geq 0$ and *CosetRep*$[u] \leq n$ then $u \notin$ *OrbitQueue*
   a. u is either previously a singleton orbit, or in an outside orbit
   b. if *CosetRep*$[u] = 0$, u was previously a singleton orbit
      - if *CosetRep*$[v] = 0$, v is also a singleton orbit $\rightarrow$ increment deltaOrbs, and assign *CosetRep*$[u]$ and *CosetRep*$[v]$
      - if *CosetRep*$[v] \neq 0$, the number of orbits doesn't change
   c. otherwise *CosetRep*$[u] \neq 0$, u is in an outside orbit. If also *CosetRep*$[v] \neq 0$, then the orbits of u and v may merge.
      - $u' :=$ FindOrbitRep(u), $v' :=$ FindOrbitRep(v)
      - if $u' \neq v'$ then merge the orbits, and decrement deltaOrbs
4. the outside orbits are now up to date with repsect to g
   we must still adjust the *Transitivity*
   a. if *Transitivity*$= 0$, G was previously an identity group
      - if deltaOrbs > 0 then *Transitivity* $= -$deltaOrbs
   b. else if *Transitivity*> 0, G was previously a transitive group, it may now be intransitive
      - if deltaOrbs > 0 then *Transitivity* $= -$deltaOrbs
   c. otherwise *Transitivity*< 0, G was previously an intransitive group, adjust the number of outside orbits
      - *Transitivity* $=$ *Transitivity* $-$ deltaOrbs
5. if *Transitivity*≥ 0, G is now a transitive group
   we must set the *Transitivity*
   a. let k be the transitivity of G_u and let m be the number of points moved by G_u
   b. if $k > 0$ and G moves $m + 1$ points, then *Transitivity* $:= k + 1$
   c. otherwise G cannot be multiply transitive.
      - if G moves only 2 points, then *Transitivity* $:= 2$
      - otherwise *Transitivity* $:= 1$

3. Correctness

The transitivity algorithm is a modified form of the Schreier-Sims algorithm. A group G is stored as a tower of stabiliser subgroups. The outside orbits of an intransitive group in the tower are stored in a merge-find data structure [4,6]. For space-efficiency, the *CosetRep* array is used to store the MF data structure, although it could be implemented using a separate array. A count of the number of outside orbits is maintained in the variable *Transitivity*. So long as *Transitivity*< 0, we know that a group is intransitive. If *Transitivity*≥ 0, the correctness of the algorithm is based

on recursion (induction). The value of *Transitivity* is set near the end of the procedure *CheckTransitivity*.

Consider the first time when *Transitivity*≥ 0 is detected. There are no outside orbits. When a stabiliser group in the tower is created and initiallized, its value of *Transitivity* is set to 0. The first time that *CheckTransitivity* encounters a value of *Transitivity*≥ 0 is at the bottom of the tower of stabilisers. This is because *CheckTransitivity* is called *after* the orbit building loop of *AddGen* completes, and *AddGen* is a recursive procedure. So assume that a generator g has been added to a group G in the tower and that *CheckTransitivity* is called, and finds a value of *Transitivity*≥ 0, indicating no outside orbits. The stabiliser G_u is either an intransitive group, or the identity group. Since G has no outside orbits, all the points moved by g are in one orbit. If G moves only two points, g is a transposition, in which case G is the symmetric group S_2. In this case, *CheckTransitivity* sets *Transitivity*$= 2$. Otherwise G moves three or more points. Its stabiliser G_u is either intransitive or the identity. In both cases, G is only 1-transitive. *CheckTransitivity* sets *Transitivity*$= 1$. These are the correct values when *CheckTransitivity* detects *Transitivity*≥ 0 for the first time.

We now proceed by induction on the number of stabiliser groups beneath G in the tower for which *CheckTransitivity* detects *Transitivity*≥ 0. We assume that when *AddGen* returns, the value stored in *Transitivity* is the correct value, when there are at most i groups in the tower beneath G for which this occurs, where $i \geq 0$. We know that it holds when $i = 0$. Consider now a situation for which *CheckTransitivity* detects *Transitivity*≥ 0. The transitivity of G_u is known to be $k \geq 0$, by the induction hypothesis. If $k > 0$ and G moves exactly one more point than G_u, then G is $(k+1)$-transitive. This is the value of *Transitivity* set by *CheckTransitivity*. Otherwise either $k = 0$ or G moves too may points. In both cases *CheckTransitivity* sets *Transitivity* to one. These are the correct values.

We can conclude that *CheckTransitivity* always sets *Transitivity* correctly. It follows that when G_u is found to be $(k-2)$-transitive or more on k points, where $k \geq 2$, that G_u is either alternating or symmetric. At this point, if G has no outside orbits, *AddGen* calls *SymmetricOrAlternating*(G, k), which checks if G moves exactly one more point than G_u. If this is the case, G is also either symmetric or alternating. The algorithm fills in the table of coset representatives, and distinguishes between alternating and symmetric groups. The orbit building loop of G is stopped at this point, since G is completely known. Hence:

3.1 Theorem. The algorithm is correct. It detects symmetric and alternating groups, and calculates the transitivity of all groups in the tower.

4. Data Structures and Performance

The data structures for storing a group are taken from [4], as is the implementation of the Schreier-Sims algorithm. A permutation is stored as an array. An array of coset representatives is stored as pointers to permutations. A group contains: a linked list of generators (permutations); an orbit stored as an array; and an array of pointers to coset representatives. It also contains a pointer to the stabiliser, which is stored in the same fashion.

In order to save storage we use the *CosetRep* array to store the outside orbits in a merge-find data structure. This works well, but it does require some subtlety in programming.

If the transitivity check is not used in the Schreier-Sims algorithm, we find that symmetric and alternating groups are much, much slower to generate than other groups. This is because of their high degree of transitivity. An n-transitive group acting on n points will require a table of $\binom{n}{2}$ coset representatives in total. Since these coset representatives are constructed by multiplying permutations, it takes at least $O(n^3)$ steps to build the table for symmetric and alternating groups. When the table is being constructed in the orbit building loop of *AddGen*, each generator is applied to every point in the orbit, and generators g for the stabiliser are constructed by multiplying three permutations. It can take up to $O(n^2)$ steps to determine whether g is already known to be in the stabiliser. So conceivably, it could take $O(n^4)$ steps for each generator to build the tables of coset representatives for the symmetric and alternating groups. If the number of generators is bounded by a constant this gives up to $O(n^4)$ steps. If the number of generators is not bounded by a constant, the complexity could become slightly worse.

When the transitivity check is used, most of the redundant generators can be avoided. As soon as G_u is found to be symmetric or alternating, the table is filled in, taking at most $O(n^3)$ steps. In fact, this step could also be avoided. Once we have discovered that a group G is either symmetric or alternating, we know its coset representatives without constructing them. So it would be enough to save a constant indicating that G is symmetric or alternating, and avoid building the table of cset representatives. The reason I have not done this is because having constructed G, it will likely be used for further computation, eg, finding Cayley graphs, finding subgroups and their cosets, etc. These other algorithms would require special programming to deal with symmetric and alternating groups if the table of coset representatives is not complete. It is easier just to build the tables of coset representatives.

It is difficult to estimate the actual complexity of the transitivity-

checking algorithm. When *SymmetricOrAlternating* is called on a group with k generators acting on n points, it can apply at most every generator to every point, a total of kn steps, which is $O(n)$ if k is bounded by a constant. It is very fast. It usually returns *true*, in which case the rest of the orbit building loop of *AddGen* is avoided. This can happen at most once at each level in the recursion, for each generator of G. In practice, the generation of symmetric and alternating groups is almost instantaneous. Previously symmetric and alternating groups were much, much slower to build, perhaps 10 times slower when $n = 24$, using the same algorithm but without the transitivity check. When G is an alternating group, the algorithm must check that each generator is even, at each level in the recursion. Since it takes n steps to check if a permutation is even, this requires an additional $O(n^2)$ steps, again assuming that the number of generators on each level is bounded by a constant. This is still very fast. It is difficult to accurately estimate the number of generators that will be produced for the stabilisers in the Schreier-Sims algorithm. In practice, the number seems to be very small.

Groups which are not symmetric or alternating take only slightly longer to build than previously. The only difference is that some time is required in order to keep track of the outside orbits. This is approximately $O(n)$ steps per generator, at each level in the recursion. The depth of the recursion will be much less than n for groups which are not alternating or symmetric. If the number of generators at each level is bounded by a constant, this will mean an additional $O(n^2)$ steps in total keeping track of outside orbits.

A multi-transitive, non-symmetric, non-alternating group can be at most 5-transitive, so that its depth of the tower of stabilisers is at most five. At each level, each generator is applied to all points in the orbit, that is, to at most n points. Multi-transitive groups will be generated in approximately $O(n^3)$ time.

The most difficult groups to build will be those that are a direct product of symmetric groups, say $S_{n/2} \times S_{n/2}$. The algorithm will correctly detect one of the symmetric groups, since it will appear in the tower of stabilisers, and build it in time $O(n^3)$. However the other will not be so easily detected, and may take $O(n^4)$ time, assuming that the number of generators on each level is bounded by a constant.

The algorithm works by detecting symmetric and alternating groups via their transitivity, and avoids the orbit building loop of the Schreier-Sims algorithm in these cases. It cannot detect representations of S_n and A_n on more than n points. For example, the group of the line graph of the complete graph K_n is isomorphic to S_n, but acts on $\binom{n}{2}$ points. The algorithm is unable to make use of the transitivity in this case, since this group is only 1-transitive. Similarly it cannot detect products of symmetric

and alternating groups, like wreath products and other subdirect products. For direct products of symmetric and alternating groups, the transitivity check is partly ineffective, since it only works when groups are transitive.

In conclusion, the complexity of the transitivity-checking Schreier-Sims algorithm is $O(n^4)$ in general, assuming that the number of generators at each level is bounded by a constant. For symmetric and alternating groups it is $O(n^3)$, with the same assumption.

Question. Can the algorithm be modified to detect direct and subdirect products involving symmetric and alternating groups?

I have used this algorithm in the graph isomorphism computation [5] of my **Groups & Graphs 2.5** software. The result has been a great increase in speed generating automorphism groups of complete graphs.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Toronto, 1974.
2. Christoff Hoffmann, *Group Theoretic Algorithms and Graph Isomorphism*, Lecture Notes in Computer Science #136, Springer-Verlag, New York, 1982.
3. D. Knuth, "Notes on efficient representation of perm groups", unpublished manuscript.
4. William Kocay, "On Writing Isomorphism Programs", book chapter in *Computational and Constructive Design Theory*, Editor: W.D. Wallis, pp. 135-175, Kluwer Academic Publishers, 1996.
5. William Kocay, "Groups & Graphs, a Macintosh application for graph theory", Journal of Combinatorial Mathematics and Combinatorial Computing 3 (1988), 195-206.
6. Mark Allen Weiss, *Data Structures and Algorithm Analysis*, Benjamin Cummings Publ. Co., Redwood City, California, 1992.