Implementations of 3 Types of the Schreier-Sims Algorithm

Martin Jaggi m.jaggi@gmx.net

MAS334 - Mathematics Computing Project

Under supervison of Dr L.H.Soicher

Queen Mary University of London

March 2005

1 Permutation Groups

1.1 Definitions ans basic properties

A *permutation group* on a set Ω is a subgroup of the *symetric group* $Sym(\Omega)$ (all permutations of the set Ω with operation composition).

If G is an arbitrary group, an *action* of G on Ω is a group homomorphism $G \longrightarrow Sym(\Omega)$. So the image of an action is a permutation group. (Sometimes actions are called *permutation representations*, and $|\Omega|$ is called the *degree* of such a representation.)

We denote the *image* of an element (or 'point') $\omega \in \Omega$ under the action of $g \in G$ by ω^g.[1]

The *orbit* of a point $\omega \in \Omega$ under G is the set $\omega^G := \{\omega^g \mid g \in G\}$. Orbits partition the set Ω and 'being in the same orbit' is an equivalence relation.

The *stabilizer* of a point ω is the subgroup $G_\omega := \{g \in G \mid \omega^g = \omega\}$.

We say that a group G is *generated* by a subset S, $G = \langle S \rangle$, if every element of G can be written as a product $s_1 \cdots s_r$ with $s_i \in S$ or $s_i^{-1} \in S$ for all i.

[1]In the C++ implementations, we will denote the image of ω under g by `g.p[`ω`]`

Proposition 1.1 (Orbits and Stabilizers). *If a finite group G acts on Ω, and $\omega \in \Omega$, then*

$$|G|/|G_\omega| = |\omega^G|.$$

Proof. We show that $f : G/G_\omega \longrightarrow \omega^G$, $G_\omega g \longmapsto \omega^g$ is a bijection:
well-defined: $G_\omega g = G_\omega h \Rightarrow G_\omega g h^{-1} = G_\omega \Rightarrow g h^{-1} \in G_\omega \Rightarrow \omega^{gh^{-1}} = \omega \Rightarrow \omega^g = \omega^h$.
surjective: Let $\omega^g \in \omega^G$. Then $f(G_\omega g) = \omega^g$.
injective: Let $\omega^g = \omega^h$. Then $\omega^{gh^{-1}} = \omega \Rightarrow gh^{-1} \in G_\omega \Rightarrow g \in G_\omega h \Rightarrow G_\omega g \subseteq G_\omega h$, and $G_\omega h \subseteq G_\omega g$ similarly, i.e. $G_\omega h = G_\omega g$. □

In the following we will assume that G is a finite group acting on Ω, $G = \langle S \rangle$ with $|S| = r$ and $|\Omega| = n$.

The above proposition leads us to a possible way to calculate the order of (big) permutation groups that are given by relatively few generators $s \in S$. We have $|G| = |\alpha^G||G_\alpha|$, so if we could calculate $|\omega^G|$ and if we could find generators for the stabilizer G_α, we would have reduced the problem to the smaller problem of finding the order of G_α. It turns out that the first part is easy to achieve, but the second part is a bit harder to do in an efficient way:

1.2 Schreier Trees and Schreier's Lemma

Schreier Trees A *Schreier tree* with root α for S is a representation of the orbit of α in the following sense:

It's a tree rooted at α with the elements of α^G as its vertices, and its edges describing the elements of S needed to get from α to each vertex, i.e. each edge $\{i, j\}$ in the tree with i closer to the root than j is labeled by a generator $s \in S$ moving i to j.

Schreier trees can be found by Breadth First Search (or Depth First Search) starting from α and applying all generators $s \in S$ trying to reach a new point α^s. So the time needed to calculate a Schreier tree is bounded by $O(rn)$. So we can find $|\alpha^G|$ in an efficient way.

The procedure `ScheierTree(int alpha)`: In my C++ implementations not the entire Schreier tree is stored, but only a coset representative t_ω for each point ω of the orbit, i.e. the product of the permutations we see as labels of the edges when we go from α to ω. These t_ω are stored in `cosrep[`ω`]`, where `cosrep[`ω`]` is set to undefined if the point ω is not in the orbit. The size of the orbit is denoted by `cosreps`.

The following important result states how a Schreier tree for α can be used to find generators for the Stabilizer G_α:

Proposition 1.2 (Schreier's Lemma). *Let $G = \langle S \rangle$. Then the stabilizer G_α of α is generated by the set of Schreier generators:*

$$G_\alpha = \langle t_i \, s \, t_{i^s}^{-1} \ \mid \ i \in \alpha^G, \, s \in S \rangle$$

where t_i is defined to be an element of G moving α to i, i.e. a coset representative of i.

Proof. We show both inclusions:

$\boxed{\supseteq}$: Clear by definition of the Schreier generators: t_i moves α to i, s moves i to i^s and $t_{i^s}^{-1}$ moves i^s back to α.

$\boxed{\subseteq}$: Let $g \in G_\alpha$. Then $g = s_1 \cdots s_r$ is a product of elements $s_i \in S$. Suppose $r > 0$ (i.e. $g \neq 1$). Let j be the maximal index such that $s_1, \ldots, s_j$ is a path in the Schreier tree with root α. Let $\beta = \alpha^{s_1 \cdots s_j}, t_\beta = s_1 \cdots s_j$. Now consider $(t_\beta \, s \, t_{\beta^s}^{-1})^{-1} g$, where $s = s_{j+1}$, and rewrite is as

$$(t_\beta \, s \, t_{\beta^s}^{-1})^{-1} g = t_{\beta^s} s^{-1} (s_1 \cdots s_j)^{-1} s_1 \cdots s_r = t_{\beta^s} s_{j+2} \cdots s_r$$

Then we apply the same reasoning to this element. Since t_{β^s} corresponds to a path in the tree from α, this procedure will end with en element t_γ in at most $r-j$ steps. However, as all elements at the left-hand-side of the equality stabilize α, the element t_γ has to be the trivial element. This implies that g is an element in the group generated by the Schreier generators. $\square$

1.3 Bases and strong generating sets

A *base* B for G is a sequence $B = (b_1, \ldots, b_k) \subseteq \Omega$ such that the pointwise stabilizer $G_{b_1,\ldots,b_k}$ is trivial.

A *strong generating set* (*SGS*) for G relative to B is a set $S \subseteq G$ such that $\langle S \cap G^{(i)} \rangle = G^{(i)}$ for each i, where $G^{(i)} := G_{b_1,\ldots,b_i}$, $G^{(0)} := G$. [3]

2 Algorithms

In the following we give the description and implementation of 3 types of the Schreier-Sims Algorithm to calculate the order of a permutation group G. As a 'by-product', these 3 algorithms are also able to compute a *base* and a *strong generating set* for G.

All programs were developed using gcc, but should also work with any other C++ compiler. All programs share the same **input format**:

> The first input is n, i.e. the size of $\Omega = \{1, \ldots, n\}$.
>
> The second input is r, the number of generators for the group.
>
> The following rn input numbers describe the r generators in image format, i.e. for each generator the image of the point 1 followed by the image of the point 2 up to the image of the point n is stated.

The $2 + rn$ input integer numbers can be separated by one or more `space` or `enter` characters, i.e. for example the permutations from the ATLAS of Finite Group Representations [4] in `Meataxe` format can be inserted directly by Copy-Paste.

2.1 Basic Schreier-Sims Algorithm

Implementation: `basic-schreier-sims.cpp`

Algorithm 1:

1. If G is non-trivial, choose a point $b \in \Omega$ that has not yet been chosen.
2. Calculate a Schreier tree with root b and obtain $|b^G|$.
3. Use Schreier's Lemma to find generators for G_b.
4. Apply this algorithm recursively for G_b, to find $|G_b|$.

If we denote the choosen points b by $b_1, \dots, b_m$ and if we write $G^{(i)}$ for $G_{b_1,\dots,b_i}$ as before, the algorithm will calculate the order of our group G as

$$|G| = |b_1^{G^{(0)}}||b_2^{G^{(1)}}| \cdots |b_m^{G^{(m-1)}}|. \tag{1}$$

The algorithm will stop as soon as $G^{(m)}$, the pointwise stabilizer of $b_1, \dots, b_m$, is the trivial group (i.e. as soon as no non-trivial element in G fixes all the points $b_1, \dots, b_m$). Obviously this will be the case after at most n steps.

The problem is that in each step, the number of generators obtained by Schreier's lemma grows by the factor $|b^G|$, so in the worst case (if each orbit is the entire Ω and it takes n steps to finish), we have rn^n generators at the end. So this algorithm needs exponential amount of memory, and so is not practicable for $n \approx 100$.

Lemma 2.1. *The running time of the Basic Schreier-Sims Algorithm is exponential.*

This makes it clear that we have to improve the algorithm in order to make it usefull in practice. In the following two sections we will give two possible ways of significant improvements.

Note that when the algorithm stops, $(b_1, \dots, b_m)$ is a *base*, and the union of all the generators we have found is a *strong generating set.*

2.2 Schreier-Sims Algorithm with Jerrum's Filter

Implementation: `sims-with-jerrums.cpp`

The following proposition makes it possible to improve the basic algorithm descibed above so that surprisingly the number of generators for $G^{(i)}$ does not increase at all during execution of the algorithm:

Proposition 2.2 (Jerrum's Filter). *[2] Any subgroup of S_n can be generated by at most $n-1$ elements. Moreover, such a generating set can be found 'on-line', in the sense that if S is a suitable generating set for H and g any element of S_n, then is is possible to find a suitable generating set S' for $\langle H, g \rangle$.*

Proof. Let $\Omega = \{1, 2, \dots, n\}$. With any non-identity permutation g of Ω, we associate an element and a 2-subset of Ω as follows:

- $i(g)$ is the smallest point of Ω moved by g;
- $e(g) = \{i(g), i(g)^g\}$.

Now, given any set S of permutations, the set $e(S) = \{e(g) | g \in S\}$ is the edge set of a graph on the vertex set Ω. We claim that

any subgroup of S_n can be generated by a subset of S such that the graph $e(S)$ contains no cycles.

This will prove the theorem, since an acyclic graph on n vertices has at most $n-1$ edges (with equality if and only if it is a tree). So suppose that $e(S)$ is acyclic and g is any permutation. We want an 'acyclic' generating set S' for $\langle S, g\rangle$. There are three cases:

- $g = 1$. Then take $S' := S$
- $e(S \cup \{g\})$ is acyclic. Take $S' := S \cup \{g\}$
- $e(S \cup \{g\})$ contains a unique cycle, which includes $e(g)$. Let $S_1 := S \cup \{g\}$. Moreover, for any set T of permutations containing a unique cycle, we let $m(T) := \sum_{g \in T} i(g)$. We show how to construct from S_1 a set S_2 with $\langle S_2\rangle = \langle S_1\rangle$ such that either $e(S_2)$ is acyclic, or $e(S_2)$ contains a unique cycle and $m(S_2) > m(S_1)$. Since $m(T)$ is bounded above for any such set (for example, by n^2, since there are at most n permutations and $i(g) \leqslant n$ for any $g \neq 1$), the second alternative can only occur finitely many times, and eventually we reach a set S' such that $\langle S'\rangle = \langle S_1\rangle$ and $e(S')$ is acyclic. Take this as the required S'.

It remains to show how to do the replacement step above.

Let C be the unique cycle in $e(S_1)$, and let i be the smallest point lying on any edge on C. Then we can travel round the cycle starting at i, recording our progress by elements g or g^{-1} according as the edge is from $i(g)$ to $i(g)^g$ or vice versa. We obtain a product $h = g_{i_1}^{\epsilon_1} \cdots g_{i_k}^{\epsilon_k}$, with $\epsilon_j = \pm 1$ for $1 \leqslant j \leqslant k$, such that h fixes i. Clearly it also fixes every point smaller than i. So, if we delete from S_i the element g_{i_1} and replace it with h, we increase the value of m (since $i(h) > i(g_{i_1}) = i$). Moreover, removing g_{i_1} produces an acyclic set, and so the addition of h at worst creates one cycle; and g_{i_1} can be expressed in terms of h and the other generators, so the groups generated by the two sets are equal.

This concludes the proof. $\square$

Now we can use this filter for the Schreier generators in our algorithm:

Algorithm 2:

1. If G is non-trivial, choose a point $b \in \Omega$ that has not yet been chosen.
2. Calculate a Schreier tree with root b and obtain $|b^G|$.
3. Use Schreier's Lemma to find generators for G_b.
4. 'Throw' the Schreier generators for G_b one-by-one into Jerrums Filter, to keep the the number of generators for G_b bounded by $n - 1$.
5. Apply this algorithm recursively for G_b, to find $|G_b|$.

Lemma 2.3. *The running time of the Schreier-Sims Algorithm with Jerrum's Filter is $O(n^7)$, i.e. polynomial in n.*

Proof. .

> **Claim:** Proceeding a permutation through Jerrum's filter takes $O(n^4)$ time.
>
> **Proof:** When we 'throw' a new generator into Jerrum's filter, the number of times we have to do a replacement step until the condition is satisfied is bounded by n^2 (see above). Doing a replacement step takes $O(n^2)$ time (at most n for finding[2] the cycle, at most n for finding it's smallest vertex, and at most n^2 for traveling i.e. multiplying around the cycle).

Now during execution of the algorithm, there at most n basepoints, and for each of the stabilizers $G^{(i)}$ there at most $n(n-1)$ Schreier generators[3] we have to throw into Jerrum's filter. So the running time of the algorithm is $O(n^3 n^4)$. □

[2]To find a cycle we do a depth-first search, visiting each vertex at most once.

[3]In case there are $r > n$ generators given at the beginning, we apply Jerrum's filter r-n times before we start the algorithm.

Note that when the algorithm stops, $(b_1, \ldots, b_m)$ is a *base*, and the union of all the generators we have found is a *strong generating set* of size $\leqslant m(n-1) = O(n^2)$.

2.3 The Incremental Schreier-Sims Algorithm

Implementation: `incremental-sims.cpp`

In this section we will look at a fast algorithm to construct a *strong generating set*. If we have a *strong generating set* for a group G, it's easy to compute the order of the G because we instantly have generators for all Stabilizers $G^{(i)}$ we need in equation (1).

A *partial base* $B = [b_1, \ldots, b_k]$, and a *partial strong generating set* S is a set $B \subseteq \Omega$ and a set $S \subseteq G$ such that no element of S fixes every element of B.

We give an algorithm that takes any *partial base* and *partial strong generating set*, and transforms it into a *base* and *strong generating set*. The following is somethimes referred as the incremental Schreier-Sims Algorithm:

Algorithm 3:

1. If $S = \{\}$ return B, S

2. At this point we have a nonempty partial base $B = (b_1, \ldots, b_k)$, and a partial strong generating set S. Set $C := [b_2, \ldots, b_k]$, $T := S \cap G_{b1}$, and apply this algorithm (recursively) with input C, T, so that they are modified to be a base and associated strong generating set for $H = \langle T \rangle$.

3. Set $B := B \cup C$, $S := S \cup T$. Now we can do membership testing in $H \leqslant G_{b_1}$, using the sifting algorithm. We test each Schreier generator s for G_{b_1} to see if $s \in H$. If all of them are in H then we have $H = G_{b_1}$ and we are done, and return B, S.

4. Otherwise we have a Schreier generator $s \in G_{b_1}$ but $s \notin H$. We set $S := S \cup \{s\}$. If s fixes all points of B, we append to B a point of Ω which is moved by s. We now go to step 2. [1]

We describe the action of $G^{(i)}$ on the cosets of $G^{(i+1)}$ by a Schreier tree we call "T_{i+1}"

The Sifting Procedure (Membership test in a group given by generators)

Suppose g is an arbitrary element of $Sym(\Omega)$. We now describe a procedure called 'sifting', which either writes g as a word in the elements of the strong generating set S for G or shows that $g \notin G$. First suppose that g fixes each base point $b_1, \ldots, b_k$. If $g = 1$ then $g \in G$ is the empty word in the strong generators, and if $g \neq 1$ then $g \notin G$. Now we may suppose that g fixes each of $b_1, \ldots, b_i$ for some $i < k$, and moves b_{i+1}. If $b_{i+1}^g \notin b_{i+1}^{G^i}$, then we conclude that $g \notin G$. Otherwise, by using the Schreier tree T_{i+1} we find elements $s_1, \ldots, s_r$ of $S \cap G^i$ such that $b_{i+1}^g = b_{i+1}^{s_1,\ldots,s_r}$. Then $h := g(s_1 \cdots s_r)^{-1}$ fixes each of $b_1, \ldots, b_{i+1}$. We may now (recursively) apply the sifting procedure to h to either determine that h, and hence g is not in G, or to find a word v in the elements of $S \cap G^{i+1}$ such that $v = h$. In the later case, $g = vs_1 \cdots s_r$ is a word in the strong generators from S. [1]

Lemma 2.4. *The running time of the Incremental Schreier-Sims Algorithm is $O(n^8 \log^3 n)$, i.e. polynomial in n.*

Proof. Let λ_i be the number of times a new generator s is added to $S \cap G^{(i)}$ during execution of the algorithm.

Every time a new generator s is added to $S \cap G^{(i)}$, the size of $\langle S \cap G^{(i)} \rangle$ at least doubles (since when $s \in G^{(i)} - H$, for every $g \in H$ we have $gs \notin H$).

Therefore $\lambda_i \leqslant \log |G^{(i)}|$ and the overall number of times a new generator s is added satisfies

$$|S| - r \;=\; \sum_{i=1}^{m} \lambda_i \;\leqslant\; \sum_{i=1}^{m} \log |G^{(i)}| \;\leqslant\; \sum_{i=1}^{m} \log |G| \;\leqslant\; n \log n! \;=\; O(n^2 \log n)$$

So at the end we have $|S| = O(n^2 \log n)$ generators[4], and for every one of these - when is is added to $S \cap G^{(i)}$ - we have to sift at most $n(\lambda_i + r) = O(n^2 \log n)$ Schreier generators[5] to see if they are in H. So there are $O(n^4 \log^2 n)$ Sifting processes.

Sifting takes at most n times the time of building a Schreier tree, since with every call of the Sifting procedure the sifted element stabilizes one more point.

[4]Supposed that the number of input generators, r, is either fixed or $O(n^2 \log n)$

[5]Supposed that the number of input generators, r, is either fixed or $O(n \log n)$

Building a Schreier tree when we have t generators is $O(n^2 + nt)$, or $O(nt)$ for $t > n$. So in our case, since we have bounded our number of (Schreier) generators t by $O(n^2 \log n)$, every Sifting process can be done in $nO(n(n^2 \log n)) = O(n^4 \log n)$. $\square$

Note that when the algorithm stops, B is a *base*, and S is a *strong generating set* of size $O(n^2 \log n)$.

The major speed-up of this algorithm compared to the basic first version of the Schreier-Sims algorithm comes from the fact that this algorithm avoids the inclusion of many redundant (schreier) generators, by only adding generators to S that are outside $\langle S \cap G^{(i)} \rangle$.

2.4 Summary

We can summarize our results for the running times and the sizes of the *strong generating sets* produced by the 3 algorithms in the following table:

Algorithm	**Running time**	**Size of produced SGS**
Basic Schreier-Sims	$O(n^n)$	$O(n^n)$
Jerrum's Filter	$O(n^7)$	$O(n^2)$
Incremental Schreier-Sims	$O(n^8 \log^3 n)$	$O(n^2 \log n)$

3 Applications

The orders of the following groups were calculated with one or both of the implementations `sims-with-jerrums.cpp` and `incremental-sims.cpp`. The last two columns state the running time in seconds, where both programs were running on a sun machine with a 500 MHz processor.

Group	$\|\boldsymbol{\Omega}\|$	$\|\mathbf{G}\|$	t Jerrums	t Inc.Sims
S_{10}	10	10!	<0.5	<0.5
S_{20}	20	20!	<0.5	1.0
S_{30}	30	30!	1.6	7.3
S_{40}	40	40!	7.6	30.2
S_{50}	50	50!	21.8	94.0
S_{60}	60	60!	54.6	243.4
S_{70}	70	70!	122.3	564.8
J_1 (Janko group)	266	175560	<0.5	13.1
McL (McLaughlin grp)	275	898128000	2.2	92.4
Co_3 (Conway group)	276	495766656000	12.9	412.0
$5^{2+2+4} : (S_3 \times GL_2(5))$ [6]	750	1125000000	47.3	-
Exeptional group ${}^2F_4(2)'$	1600	17971200	22.3	-
Suz (Suzuki group)	1782	448345497600	some time	-

The test with the 'full' permutations groups S_n was made using a single transposition and a left shift as generators for the group[7], e.i. the input was

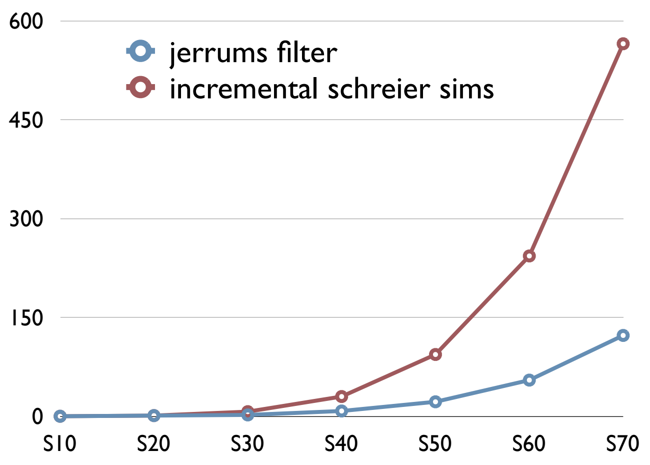
```
n
2
2 1 3 4 ...n
n 1 2 ... n-1
```

Two generators for each of the other groups were taken from [4].

[6][4] states that this is one of the maximal subgroups of the Monster group M

[7]using the fact that every transposition can be written as a product of these two permutations, and that every permutation can be written as a product of transpositions.

Comparison of the two algorithms for input S_n:



Running times in seconds to calculate the order $(n!)$ of S_n for $n = 10, 20, \ldots, 70$.

If we make the very simplified proposal for the running time that $t = cn^p$, we can determine a very experimental 'degree' of the running time polynomial, given two measured runs:

$$t_1 = cn_1^p \ , \ t_2 = cn_2^p \quad \Rightarrow \quad p = \frac{\log \frac{t_2}{t_1}}{\log \frac{n_2}{n_1}}$$

For $n_1 = 60, n_2 = 70$ and input S_n we get a degree for `sims-with-jerrums.cpp` of 5.23 compared to 5.46 for `incremental-sims.cpp`.

4 Appendix - Some remarks about the programs

The input format for all implementations was already stated above. For storing and calculating with permutations, the 3 programs make use of the class `Permutation`. The class stores the permutations in image format, i.e. the image of each point i is stored in `g.p[`i`]` [8]. The class allows multiplication of two permutations; `g*h` becomes gh where the products are read from left to right. It further allows inversion (`g.inverse()`), comparison (`g == h`), input (`g.input()`) and output (`g.output()`) of permutations, each of these elementary functions taking time $O(n)$.

4.1 `basic-schreier-sims.cpp`

The procedure `void ScheierTree(int` *alpha*`)` performes a depth-first search to determine the orbit of $alpha \in \{0, \ldots, n-1\}$ under the group generated by the actual generators `g`. For each orbit point i that it finds, it stores a coset representative in `cosrep[`i`]`, i.e. a permutation that maps *alpha* to i. Otherwise `cosrep[`i`]` remains the undefined permutation.

The procedure `void SchreierSims()` implements **Algorithm 1** by choosing new base points *alpha* as long as the group $\langle$`g`$\rangle$ is non-trivial. In each step it calculates the orbit of *alpha* (`ScheierTree(`*alpha*`)`) and then applies Schreier's lemma to new get generators `newg` for G_{alpha}. Finally it replace the generators `g` by these new generators, ready to call the algorithm again. This implementation will only add Schreier generators that did not occur yet, but this unfortunately does not mean they are not redundant.

4.2 `sims-with-jerrums.cpp`

This implements **Algorithm 2**, and uses the same basic structure as `basic-schreier-sims.cpp`, but with addition of Jerrum's filter.
`void JerrumsFilter()` actually calculates the same Schreier generators as described before, but instead of storing them, each of them is getting 'thrown' into Jerrum's filter. This is realized by `void ThrowIn(Permutation` g`)`.

[8] before storage, the points $\{1, \ldots, n\}$ are mapped onto $\{0, \ldots, n-1\}$ for making them usable in C/C++ arrays.

`JVertex[`i`]` represents the vertex of Jerrum's graph corresponding to point i. It has a list `neighbor` of it's neigbor vertices, and a list `isneighbor`, where if j is a neighbor of i, `isneighbor[`j`]` points to the index of j in the `neighbor` list, and otherwise is set to -1. Additionally a boolean variable `visited` is used by the `FindCycle` procedure when searching for a cycle in the graph.

The program `sims-with-jerrums-n^3.cpp` stores a permutation `perm` for every possible neighbor for each vertex in Jerrum's graph, which results in using memory for n^3 integers, but making the program a bit easier to read. The improved progam `sims-with-jerrums.cpp` only keeps a separate list `jg` of the permutations that are really edges of Jerrum's graph, and so only uses $O(n^2)$ in space.

4.3 `incremental-sims.cpp`

This program implements **Algorithm 3**, where `basepoint` is used to store the *partial base* B and `sgs` is used to store S, the *partial strong generating set*. There is a small modification to the `Permutation` data structure used in the programs above: for each permutation g the first basepoint moved by g is stored in `g.fbp`. This makes it much faster to get access to the sets $S \cap G^{(i)}$ that we use during the algorithm.

The boolean function `bool Sifting(Permutation` h`)` implements the Sifting procedure described in section 2.3, i.e. returns if $h \in \langle S \cap G^{(i)} \rangle$ or not.

After execution of `SchreierSims()`, `sgs` contains a *strong generating set* for the group G, and the order of G is then simply calculated using this *SGS*.

References

[1] A. M. Cohen, H. Cuypers, H. Sterk (1999), *Some Tapas of Computer Algebra*, Springer Verlag, pp 184-194

[2] P. J. Cameron (1999), *Permutation Groups*, London Math. Soc. Student Texts **45**, Cambridge University Press, Sections 1.13 and 1.14

[3] Á. Seress (2003), *Permutation Group Algorithms*, Cambridge Tracts in Mathematics, Cambridge University Press, pp 55-62

[4] Conway, Curtis, Norton, Parker, Wilson (1985), *ATLAS of Finite Group Representations*, http://web.mat.bham.ac.uk/atlas