Permutation Group Algorithms, Part 1

Jason B. Hill

University of Colorado

28 September, 2010

Slide one of this presentation by Jason B. Hill on polynomial time permutation group algorithms has a sentence containing ten a's, three b's, three c's, three d's, forty-one e's, nine f's, eight g's, fifteen h's, twenty-five i's, two j's, one k, eight l's, five m's, twenty-eight n's, twenty-one o's, five p's, one q, twelve r's, thirty-two s's, thirty-five t's, three u's, six v's, eight w's, two x's, nine y's, and one z.

Resources:

- GAP code for Schreier-Sims functions (under this talk) at

  `http://math.jasonbhill.com/talks`

- Alexander Hulpke's "Notes on Computational Group Theory"
- Holt, et al's "Handbook of Computational Group Theory"
- Seress' "Permutation Group Algorithms"

1. GAP and Sage

2. Background

3. Memory and Time

4. Permutation Group Algorithms in Polynomial Time

GAP and Sage

During this talk, I will reference two (free) software packages:

GAP – Groups, Algorithms and Programming

`http://www.gap-system.org/`

Sage (especially Sage Combinat)

`http://www.sagemath.org/`

`http://wiki.sagemath.org/combinat`

GAP and Sage

To create a permutation group in GAP or Sage:

GAP and Sage

To create a permutation group in GAP or Sage:

GAP: (as cycles in the GAP language)

```
gap> G:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> Order(G);
6
```

GAP and Sage

To create a permutation group in GAP or Sage:

GAP: (as cycles in the GAP language)

```
gap> G:=Group((1,2,3),(1,2));
Group([ (1,2,3), (1,2) ])
gap> Order(G);
6
```

Sage: (as a Python list of Python tuples)

```
sage:  G=PermutationGroup([(1,2,3),(1,2)])
sage:  G.order()
6
```

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.
- Unfortunately, there isn't consistency between GAP and Sage.

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.
- Unfortunately, there isn't consistency between GAP and Sage.

Example:

```
gap> G:=Group((45,46),(96,97));; DegreeAction(G);
4
```

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.
- Unfortunately, there isn't consistency between GAP and Sage.

Example:

```
gap> G:=Group((45,46),(96,97));; DegreeAction(G);
4
```

```
sage:  G=PermutationGroup([(45,46),(96,97)])
sage:  G.degree()
97
```

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.
- Unfortunately, there isn't consistency between GAP and Sage.

Example:

```
gap> G:=Group((45,46),(96,97));; DegreeAction(G);
4
```

```
sage:  G=PermutationGroup([(45,46),(96,97)])
sage:  G.degree()
97
```

- Notice that this makes a notion of primitivity in Sage, for instance, not precise with existing literature. (What is the size of a block?)

GAP –vs– Sage

- Sage uses GAP for most permutation group calculations.
- Unfortunately, there isn't consistency between GAP and Sage.

Example:

```
gap> G:=Group((45,46),(96,97));; DegreeAction(G);
4
```

```
sage:  G=PermutationGroup([(45,46),(96,97)])
sage:  G.degree()
97
```

- Notice that this makes a notion of primitivity in Sage, for instance, not precise with existing literature. (What is the size of a block?)
- Recent patches to Sage Combinat by Mike Hansen and myself have made Sage more consistent with GAP, but GAP is still better suited to serious work.

Background

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

For Permutation Groups:

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

For Permutation Groups:

- We *really* define a group along with an action.

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

For Permutation Groups:

- We *really* define a group along with an action.
- Group elements are permutations acting on Ω (usually $\Omega \subset \mathbb{Z}_{\geq 1}$).

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

For Permutation Groups:

- We *really* define a group along with an action.
- Group elements are permutations acting on Ω (usually $\Omega \subset \mathbb{Z}_{\geq 1}$).
- The group operation is composition of permutations.

Group Actions

Group Action

A group G acts on a set (domain) Ω if

- $\omega^1 = \omega$ for all $\omega \in \Omega$.
- $(\omega^g)^h = \omega^{gh}$ for all $\omega \in \Omega$ and all $g, h \in G$.

For Permutation Groups:

- We *really* define a group along with an action.
- Group elements are permutations acting on Ω (usually $\Omega \subset \mathbb{Z}_{\geq 1}$).
- The group operation is composition of permutations.
- The same group may be used in vastly different actions. Upon fixing an action, we will refer to both the group and the action as G. (We will see an example shortly of why such things matter.)

Orbits and Point Stabilizers

Orbits and Point Stabilizers

For $\omega \in \Omega$ define the orbit

$$\omega^G = \{\omega^g \mid g \in G\} \subset \Omega$$

and the point stabilizer

$$G_\omega = \mathrm{Stab}_G(\omega) = \{g \in G \mid \omega^g = \omega\}.$$

Orbits and Point Stabilizers

Orbits and Point Stabilizers

For $\omega \in \Omega$ define the orbit

$$\omega^G = \{\omega^g \mid g \in G\} \subset \Omega$$

and the point stabilizer

$$G_\omega = \mathrm{Stab}_G(\omega) = \{g \in G \mid \omega^g = \omega\}.$$

Orbit-Stabilizer Theorem: For $\omega \in \Omega$ there is a bijection between ω^G and the set $G_\omega{}^{\backslash G}$ (right cosets of G_ω in G) given by $\omega^g \leftrightarrow G_\omega \cdot g$. In particular (for finite groups anyway) $|\omega^G| = [G : G_\omega]$.

More Definitions

For a permutation group G acting on $\Omega \ldots$

More Definitions

For a permutation group G acting on Ω...

Degree

The degree of G is $|\Omega|$. Unless otherwise indicated, we set $n := |\Omega|$.

More Definitions

For a permutation group G acting on Ω...

Degree

The degree of G is $|\Omega|$. Unless otherwise indicated, we set $n := |\Omega|$.

Transitive

G is transitive if for all $\omega_1, \omega_2 \in \Omega$ there exists some $g \in G$ such that $\omega_1^g = \omega_2$.

More Definitions

For a permutation group G acting on Ω...

Degree

The degree of G is $|\Omega|$. Unless otherwise indicated, we set $n := |\Omega|$.

Transitive

G is transitive if for all $\omega_1, \omega_2 \in \Omega$ there exists some $g \in G$ such that $\omega_1^g = \omega_2$.

Primitive

G is primitive if it preserves no non-trivial equivalence relation on Ω. Equivalently, G is primitive if Ω cannot be partitioned into non-trivial “blocks” such that the action of G preserves the blocks.

More Definitions

For a permutation group G acting on Ω...

Degree

The degree of G is $|\Omega|$. Unless otherwise indicated, we set $n := |\Omega|$.

Transitive

G is transitive if for all $\omega_1, \omega_2 \in \Omega$ there exists some $g \in G$ such that $\omega_1^g = \omega_2$.

Primitive

G is primitive if it preserves no non-trivial equivalence relation on Ω. Equivalently, G is primitive if Ω cannot be partitioned into non-trivial "blocks" such that the action of G preserves the blocks.

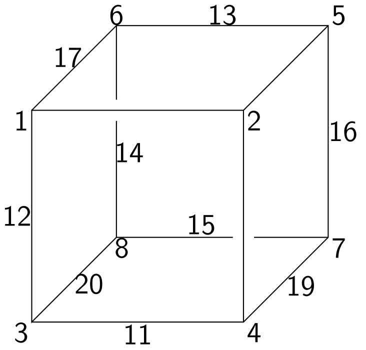
Base

A base for G is a subset $B \subset \Omega$ such that the only element of G stabilizing all points of B is the identity.

Cube Rotations

Example: Consider the rotational group of the cube:

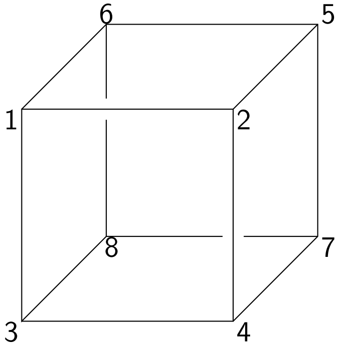
```
gap> x:=(1,2,4,3)(5,7,8,6)(9,10,11,12)(13,16,15,14)(17,18,19,20);;
gap> y:=(1,6,8,3)(2,5,7,4)(9,13,15,11)(10,18,16,19)(12,17,14,20);;
gap> z:=(1,2,5,6)(3,4,7,8)(9,18,13,17)(10,16,14,12)(11,19,15,20);;
gap> G:=Group(x,y,z);;
```



Cube Rotations

We could consider acting only on the vertices:

```
gap> r:=(1,2,4,3)(5,7,8,6);;
gap> s:=(1,6,8,3)(2,5,7,4);;
gap> t:=(1,2,5,6)(3,4,7,8);;
gap> H:=Group(r,s,t);;
```



Cube Rotations

```
gap> IsomorphismGroups(G,SymmetricGroup(4));
gap> IsomorphismGroups(H,SymmetricGroup(4));
```

show us that

$$\varphi_1 : G \to S_4 : \begin{cases} x \mapsto (1,2,3,4) \\ z \mapsto (1,4,2,3) \end{cases} \quad \text{and} \quad \varphi_2 : H \to S_4 : \begin{cases} r \mapsto (1,2,3,4) \\ t \mapsto (1,4,2,3) \end{cases}$$

are isomorphisms. Hence, $G \simeq H \simeq S_4$ while the corresponding actions are clearly different.

Cube Rotations Summary

	$G \leq S_{20}$	$H \leq S_8$	S_4
domain	$[1 \ldots 20]$	$[1 \ldots 8]$	$[1 \ldots 4]$
transitive	no	yes	yes
orbits	$[1 \ldots 8]$ and $[9 \ldots 20]$	$[1 \ldots 8]$	$[1 \ldots 4]$
primitive	no	no	yes
block system	edges and vertices	long diagonals	trivial
base	$[1, 2]$	$[1, 2]$	$[1, 2, 3]$

Cube Rotations Summary

	$G \leq S_{20}$	$H \leq S_8$	S_4
domain	$[1 \ldots 20]$	$[1 \ldots 8]$	$[1 \ldots 4]$
transitive	no	yes	yes
orbits	$[1 \ldots 8]$ and $[9 \ldots 20]$	$[1 \ldots 8]$	$[1 \ldots 4]$
primitive	no	no	yes
block system	edges and vertices	long diagonals	trivial
base	$[1, 2]$	$[1, 2]$	$[1, 2, 3]$

Moral: Intuitive understandings of S_4 can differ drastically from its real-world implementations. Such situations need to be taken into account when designing data structures or algorithms that handle these groups. If we want to work with groups having degree $> 10^6$, for instance, then intuition probably won't help guide us much. But, if we work with a well-known group, the output should make sense.

Memory and Time

Memory and Time

While our intuitive understanding of groups/actions may place extraneous requirements on a data structure or algorithm, there are two restrictions that mechanically limit even a perfect data structure or algorithm design.

1. Memory
2. Computation Time

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

- Try to store all elements. But, how?

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

- Try to store all elements. But, how?
- We know each permutation group has a base. In fact, each permutation group element induces a unique “base image” by acting on each element of the base. (Perhaps one should think of this as a reduced permutation diagram.)

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

- Try to store all elements. But, how?
- We know each permutation group has a base. In fact, each permutation group element induces a unique "base image" by acting on each element of the base. (Perhaps one should think of this as a reduced permutation diagram.)
- Thus, storing base images may be a reasonable approach.

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

- Try to store all elements. But, how?
- We know each permutation group has a base. In fact, each permutation group element induces a unique “base image” by acting on each element of the base. (Perhaps one should think of this as a reduced permutation diagram.)
- Thus, storing base images may be a reasonable approach.
- For example, in S_5 with base $[1, 2, 3, 4]$, the permutation $(1, 2)$ would be recorded as $[2, 1, 3, 4]$ and $(1, 5)(2, 3, 4)$ as $[5, 3, 4, 2]$.

Memory

How should we store (in a computer) permutation group elements?

Example of How We Could Fail:

- Try to store all elements. But, how?
- We know each permutation group has a base. In fact, each permutation group element induces a unique "base image" by acting on each element of the base. (Perhaps one should think of this as a reduced permutation diagram.)
- Thus, storing base images may be a reasonable approach.
- For example, in S_5 with base $[1, 2, 3, 4]$, the permutation $(1, 2)$ would be recorded as $[2, 1, 3, 4]$ and $(1, 5)(2, 3, 4)$ as $[5, 3, 4, 2]$.
- Limit parentheses, extra characters, etc.

Memory

How much memory does it take to store *all* elements of S_n as base images?

Memory

How much memory does it take to store *all* elements of S_n as base images?

- A base for S_n is $[1, \ldots, n-1]$, so each base image is expressed in

$$\sum_{j=1}^{n-1} \lceil \log_2(j) \rceil$$

  bits. (This is of course a conservative estimate.) Multiplying by $n!$ and dividing by 8×10^{12} we find that all elements of S_n are recordable in no less than

$$\frac{\log_2\left((n-1)!^{n!}\right)}{8 \times 10^{12}} = \frac{\log_2\left(\Gamma(n)^{\Gamma(n+1)}\right)}{8 \times 10^{12}} \text{ terabytes.}$$

Memory

How much memory does it take to store *all* elements of S_n as base images?

- A base for S_n is $[1, \ldots, n-1]$, so each base image is expressed in

  $$\sum_{j=1}^{n-1} \lceil \log_2(j) \rceil$$

  bits. (This is of course a conservative estimate.) Multiplying by $n!$ and dividing by 8×10^{12} we find that all elements of S_n are recordable in no less than

  $$\frac{\log_2\left((n-1)!^{n!}\right)}{8 \times 10^{12}} = \frac{\log_2\left(\Gamma(n)^{\Gamma(n+1)}\right)}{8 \times 10^{12}} \text{ terabytes.}$$

- In this manner, turning every atom in the observable universe into a 1TB hard disk allows one to only consider up to S_{64}.

Memory

How much memory does it take to store *all* elements of S_n as base images?

- A base for S_n is $[1, \ldots, n-1]$, so each base image is expressed in

  $$\sum_{j=1}^{n-1} \lceil \log_2(j) \rceil$$

  bits. (This is of course a conservative estimate.) Multiplying by $n!$ and dividing by 8×10^{12} we find that all elements of S_n are recordable in no less than

  $$\frac{\log_2\left((n-1)!^{n!}\right)}{8 \times 10^{12}} = \frac{\log_2\left(\Gamma(n)^{\Gamma(n+1)}\right)}{8 \times 10^{12}} \text{ terabytes.}$$

- In this manner, turning every atom in the observable universe into a 1TB hard disk allows one to only consider up to S_{64}.
- Clearly we need a different approach.

Polynomial Time

$$L \subseteq AL = P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

$$L \subsetneq PSPACE, \; P \subsetneq EXPTIME$$

Polynomial Time

$$L \subseteq AL = P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

$$L \subsetneq PSPACE, \; P \subsetneq EXPTIME$$

Polynomial Time P

An algorithm is said to be polynomial time if its running time is upper bounded by a polynomial in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant k.

Polynomial Time

$$L \subseteq AL = P \subseteq NP \subseteq PSPACE \subseteq EXPTIME$$

$$L \subsetneq PSPACE, \; P \subsetneq EXPTIME$$

Polynomial Time P

An algorithm is said to be polynomial time if its running time is upper bounded by a polynomial in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant k.

Examples:

- The "quicksort" sorting algorithm on n integers performs at most An^2 operations for some constant A. Thus it runs in time $O(n^2)$.
- All basic arithmetic operations on a computer can be done in polynomial time.

Permutation Group Algorithms in Polynomial Time

"Plain Vanilla" Orbit Algorithm

Let $G = \langle \underline{g} \rangle$ with $\underline{g} = \{g_1, \ldots, g_m\}$, acting on Ω.

“Plain Vanilla” Orbit Algorithm

Let $G = \langle \underline{g} \rangle$ with $\underline{g} = \{g_1, \ldots, g_m\}$, acting on Ω.

In some sense, the most basic algorithm is the orbit algorithm.

“Plain Vanilla” Orbit Algorithm

Let $G = \langle \underline{g} \rangle$ with $\underline{g} = \{g_1, \ldots, g_m\}$, acting on Ω.

In some sense, the most basic algorithm is the orbit algorithm.

“Plain Vanilla” Orbit Algorithm

Input: $\underline{g} = \{g_1, \ldots, g_m\}$, $\omega \in \Omega$. **Output:** ω^G.

```
$\Delta := [\omega]$;
for $\delta \in \Delta$ do
    for $i \in \{1, \ldots, m\}$ do
        $\gamma := \delta^{g_i}$;
        if $\gamma \notin \Delta$ then
            Append $\gamma$ to $\Delta$;
        fi;
    od;
od;
return $\Delta$;
```

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

- m generators of G and $|\Omega| = n$ yields a runtime of $O(nm)$.

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

- m generators of G and $|\Omega| = n$ yields a runtime of $O(nm)$.
- Thus, naively, the algorithm has a linear runtime.

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

- m generators of G and $|\Omega| = n$ yields a runtime of $O(nm)$.
- Thus, naively, the algorithm has a linear runtime.
- However, note that step 2 is non-primitive recursive.

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

- m generators of G and $|\Omega| = n$ yields a runtime of $O(nm)$.
- Thus, naively, the algorithm has a linear runtime.
- However, note that step 2 is non-primitive recursive.
- Also, step 5 is a search problem.

“Plain Vanilla” Orbit Algorithm

Example: Using the group $G \leq S_{20}$ (cube rotations) from earlier:

```
gap> jbhOrbitAlgVanilla([x,y,z],2);
[ 2, 5, 4, 6, 7, 3, 1, 8 ]
```

Some notes on the Plain Vanilla Orbit Algorithm:

- m generators of G and $|\Omega| = n$ yields a runtime of $O(nm)$.
- Thus, naively, the algorithm has a linear runtime.
- However, note that step 2 is non-primitive recursive.
- Also, step 5 is a search problem.
- Once the orbit is created, information about specifically how it was created is lost. This would be nice information to store and introduces little complexity.

Modified Orbit Algorithm

We will change steps 1 and 6 of the algorithm:

Orbit Algorithm

Input: $\underline{g} = \{g_1, \ldots, g_m\}$, $\omega \in \Omega$. **Output:** ω^G, transversal T.

```
1: $\Delta := [\omega]$, $T = [()]$;
2: for $\delta \in \Delta$ do
3:     for $i \in \{1, \ldots, m\}$ do
4:         $\gamma := \delta^{g_i}$;
5:         if $\gamma \notin \Delta$ then
6:             Append $\gamma$ to $\Delta$, Append $T[\delta] \cdot g_i$ to $T$;
7:         fi;
8:     od;
9: od;
10: return $\Delta$;
```

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T**:**

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T:

- T provides a relation between $\delta \in \omega^G$ and $g \in G$ such that $\omega^g = \delta$.

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T:

- T provides a relation between $\delta \in \omega^G$ and $g \in G$ such that $\omega^g = \delta$.
- I.e., it gives a representative in G for each orbit element.

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T:

- T provides a relation between $\delta \in \omega^G$ and $g \in G$ such that $\omega^g = \delta$.
- I.e., it gives a representative in G for each orbit element.
- A list of such representatives T is called a transversal.

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T:

- T provides a relation between $\delta \in \omega^G$ and $g \in G$ such that $\omega^g = \delta$.
- I.e., it gives a representative in G for each orbit element.
- A list of such representatives T is called a transversal.
- By the Orbit-Stabilizer Theorem, T is simultaneously a set of representatives for the cosets of $G_\omega = \mathrm{Stab}_G(\omega) \leq G$.

Modified Orbit Algorithm

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
```

About T:

- T provides a relation between $\delta \in \omega^G$ and $g \in G$ such that $\omega^g = \delta$.
- I.e., it gives a representative in G for each orbit element.
- A list of such representatives T is called a transversal.
- By the Orbit-Stabilizer Theorem, T is simultaneously a set of representatives for the cosets of $G_\omega = \mathrm{Stab}_G(\omega) \leq G$.
- Almost for free, we get limited information about stabilizer subgroups. We should exploit this idea more.

Schreier Vectors

There are some problems with this approach:

Schreier Vectors

There are some problems with this approach:

- Storing T will become a problem for long orbits.

Schreier Vectors

There are some problems with this approach:

- Storing T will become a problem for long orbits.
- Instead, we should store only pointers to generators.

Schreier Vectors

There are some problems with this approach:

- Storing T will become a problem for long orbits.
- Instead, we should store only pointers to generators.
- This is accomplished with the following definition.

Schreier Vectors

There are some problems with this approach:

- Storing T will become a problem for long orbits.
- Instead, we should store only pointers to generators.
- This is accomplished with the following definition.

Schreier Vector for $\Delta = \omega^G$

A Schreier vector is a list $S = [S[\Delta[1]], S[\Delta[2]], \ldots, S[\Delta[|\Delta|]]]$ satisfying

1. $S[\Delta[i]] \in \underline{g}$ for $1 \leq i \leq |\Delta|$.
2. $S[\omega] = ()$
3. $S[\delta] = g$ ad $\delta^{g^{-1}} = \gamma$, then γ precedes δ in Δ.

Sometimes a Schreier vector is called a “factored transversal.”

Orbit Algorithm with Schreier Vector

Orbit Algorithm with Schreier Vector

Input: $\underline{g} = \{g_1, \dots, g_m\}$, $\omega \in \Omega$. **Output:** ω^G, Schreier vector S.

```
$\Delta := [\omega]$, $S = [1]$;
for $\delta \in \Delta$ do
    for $i \in \{1, \dots, m\}$ do
        $\gamma := \delta^{g_i}$;
        if $\gamma \notin \Delta$ then
            Append $\gamma$ to $\Delta$, Append $i$ to $S$;
        fi;
    od;
od;
return $\Delta$;
```

Schreier Vectors

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
gap> jbhOrbitAlg([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), 1, 1 ] ]
```

Schreier Vectors

Example:

```
gap> jbhOrbitAlgTrans([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), (1,2,3), (1,2,3)(1,2,3) ] ]
gap> jbhOrbitAlg([(1,2,3),(4,5,6)],2);
[ [ 2, 3, 1 ], [ (), 1, 1 ] ]
```

Question: We now have a way to find coset representatives of point stabilizers in linear time. Can this be translated in some way to yield generators for the point stabilizers? (Yes.)

Schreier's Theorem

Schreier's Theorem

Let $G = \langle \underline{g} \rangle$ be a finitely generated group and $H \leq G$ with $[G : H] < \infty$. Suppose $\underline{r} = \{r_1 = 1, r_2, \ldots, r_m\}$ is a set of (right) coset representatives for H in G. For $k \in G$ write $\overline{k}$ to denote the representative $\overline{k} := r_i$ with $Hr_i = Hk$. Let

$$U := \{r_i g_j \overline{(r_i g_j)}^{-1} \mid r_i \in \underline{r},\ g_j \in \underline{g}\}.$$

Then $H = \langle U \rangle$. U is called a set of Schreier generators for H.

Schreier's Theorem

Schreier's Theorem

Let $G = \langle \underline{g} \rangle$ be a finitely generated group and $H \leq G$ with $[G : H] < \infty$. Suppose $\underline{r} = \{r_1 = 1, r_2, \ldots, r_m\}$ is a set of (right) coset representatives for H in G. For $k \in G$ write $\overline{k}$ to denote the representative $\overline{k} := r_i$ with $Hr_i = Hk$. Let

$$U := \{r_i g_j \overline{(r_i g_j)}^{-1} \mid r_i \in \underline{r},\ g_j \in \underline{g}\}.$$

Then $H = \langle U \rangle$. U is called a set of Schreier generators for H.

As a consequence, we could rewrite the orbit algorithm a final time so that it produces an orbit, a Schreier vector, and stabilizer generators.

Schreier's Theorem

Schreier's Theorem

Let $G = \langle \underline{g} \rangle$ be a finitely generated group and $H \leq G$ with $[G : H] < \infty$. Suppose $\underline{r} = \{r_1 = 1, r_2, \ldots, r_m\}$ is a set of (right) coset representatives for H in G. For $k \in G$ write $\overline{k}$ to denote the representative $\overline{k} := r_i$ with $Hr_i = Hk$. Let

$$U := \{r_i g_j \overline{(r_i g_j)}^{-1} \mid r_i \in \underline{r},\ g_j \in \underline{g}\}.$$

Then $H = \langle U \rangle$. U is called a set of Schreier generators for H.

As a consequence, we could rewrite the orbit algorithm a final time so that it produces an orbit, a Schreier vector, and stabilizer generators.

Question: Can we use this process to decompose an entire group, recursively by one point-stabilizer at a time? (Yes.)

A Slight Detour: Normal Closure

Normal Closure

Let $U \leq G$. The normal closure of U in G is

$$\langle U \rangle_G = \bigcap \{ N \mid U \leq N \lhd G \}$$

and is the smallest normal subgroup of G containing U.

A Slight Detour: Normal Closure

Normal Closure

Let $U \leq G$. The normal closure of U in G is

$$\langle U \rangle_G = \bigcap \{ N \mid U \leq N \lhd G \}$$

and is the smallest normal subgroup of G containing U.

Without too much work, the orbit algorithm can be modified to calculate normal closure. (Instead of acting on domain points by permutations, we act on group elements by conjugation.)

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

- Normal closure of subgroups.

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

- Normal closure of subgroups.
- Testing if a subgroup is normal. ($N \lhd G$ implies $\langle N \rangle_G = N$.)

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

- Normal closure of subgroups.
- Testing if a subgroup is normal. ($N \lhd G$ implies $\langle N \rangle_G = N$.)
- Computation of commutator subgroups:

$$G' = \left\langle a^{-1} b^{-1} ab \mid a, b \in \underline{g} \right\rangle_G$$

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

- Normal closure of subgroups.
- Testing if a subgroup is normal. ($N \lhd G$ implies $\langle N \rangle_G = N$.)
- Computation of commutator subgroups:

$$G' = \left\langle a^{-1}b^{-1}ab \mid a, b \in \underline{g} \right\rangle_G$$

- Computation of the derived series and lower central series.

A Slight Detour: Normal Closure

As a result, the following calculations are known to be in polynomial time:

- Normal closure of subgroups.
- Testing if a subgroup is normal. ($N \lhd G$ implies $\langle N \rangle_G = N$.)
- Computation of commutator subgroups:

$$G' = \left\langle a^{-1} b^{-1} ab \mid a, b \in \underline{g} \right\rangle_G$$

- Computation of the derived series and lower central series.
- Is G solvable? Is G nilpotent?

Thank You