Permutation Group Algorithms

Zoltán Halasi

Eötvös Loránd University

2016

Some basic algorithms for groups

Main areas of Computational Group Theory

- Permutation groups
- Matrix groups
- Finitely presented groups
- Polycyclic groups
- Group representations

CAS, References

- General Computer Algebra Systems:
  - Gap (`http://www.gap-system.org/`; free)
  - Magma (`http://magma.maths.usyd.edu.au`; to institutions for charge)
- References:
  - D. F. Holt, B. Eick, E. O'Brien: Handbook of computational group theory
  - A. Hulpke: Notes on Computational Group Theory (lecture notes)
  - Ákos Seress: Permutation Group Algorithms

Groups

Group: $(G, *)$ is a group, if G is a set and
$* : G \times G \to G,\ (a, b) \to a * b$ is a binary operation satisfying

1. Associativity: $(a * b) * c = a * (b * c)$;
2. Unit element: $\exists e \in G$ such that $e * a = a * e = a\ \forall a \in G$
3. Inverse: $\forall a \in G,\ \exists b \in G$ such that $a * b = b * a = e$.

Remarks:

- Every group is finite! (In this lecture, of course)
- Notation: $a * b \Rightarrow ab$, Unit element: 1, Inverse: a^{-1}
- Unit element and inverse are unique;
- Cancellation laws:
  $\forall a, x, y \in G,\ ax = ay \iff x = y \iff xa = ya$
- Solving equations:
  $ax = b \iff x = a^{-1}b,\ xa = b \iff x = ba^{-1}$
- Powers, power identities
- Order of an element, $o(g)$.

Subgroups, cosets

- $H \leq G$ is a subgroup if $a, b \in H \Rightarrow a^{-1}, ab \in H$;
  (If $|H| < \infty$, then $a^{-1} = a^{o(a)-1}$)
- Generated subgroup: $X \subseteq G \Rightarrow \langle X \rangle$ is the unique smallest subgroup containing X, i.e. $X \subseteq H \leq G \Rightarrow \langle X \rangle \leq H \leq G$.
- $\langle X \rangle = \{x_1^{\varepsilon_1} x_2^{\varepsilon_1} \cdots x_s^{\varepsilon_s} \,|\, s \in \mathbb{N},\ \forall 1 \leq i \leq s : x_i \in X,\ \varepsilon_i \in \{\pm 1\}\}$
- Special case: $g \in G \Rightarrow \langle g \rangle = \{g^k \,|\, 0 \leq k < o(g)\}$ is the cyclic subgroup generated by g.
- Cosets: $H \leq G$, $g \in G \Rightarrow$:
  - Left coset: $gH := \{gh \,|\, h \in H\}$
  - Right coset: $Hg := \{hg \,|\, h \in H\}$

  Terminology: Left/Right coset of H in G represented by g.
  We use right cosets from now on!

Lagrange theorem, index, transversal

- $H \leq G,\ x, y \in G \Rightarrow Hx = Hy$ or $Hx \cap Hy = \emptyset$;
- G is partitioned into right cosets of H
- The index of H in $|G|$ is $|G : H|$=the number of different (right) cosets;
- $T = \{g_1, \dots, g_k\}$ (where $|G : H| = k$) is a transversal for H in G if the list $Hg_1, \dots Hg_k$ contains each coset of H exactly once; We also say $T = \{g_1, \dots, g_k\}$ is a complete set of coset representatives;
- $\forall i :\ |Hg_i| = |H| \Rightarrow |G| = |H| \cdot |G : H|$;
- $H \leq G \Rightarrow |H| \mid |G|$. In particular, $o(g) = |\langle g \rangle| \mid |G|$ for any $g \in G$.

Permutation groups and group actions

- The symmetric group:
  Ω is a finite set, $\mathrm{Sym}(\Omega) :=$ All $\Omega \mapsto \Omega$ bijections.
  Group operation: composition of functions
- Usually, $\Omega = \{1, 2, \ldots, n\}, \Rightarrow \mathrm{Sym}(\Omega) = S_n$;
- Permutation group on Ω: $G \leq \mathrm{Sym}(\Omega)$.
- G acts on Ω if $\forall g, h \in G,\ \forall \omega \in \Omega$
  - $\exists \omega^g \in \Omega$; (The image of ω under G)
  - $(\omega^g)^h = \omega^{gh}$;
  - $\omega^1 = \omega$.
- Group action $\iff$ $G \to \mathrm{Sym}(\Omega)$ homomorphism (product presserving map).

$$g \in G \to \begin{pmatrix} \omega_1 & \omega_2 & \ldots & \omega_n \\ \omega_1^g & \omega_2^g & \ldots & \omega_n^g \end{pmatrix}$$

Some important actions

- Action on cosets:
  - Right: $H \le G,\ \Omega := \{Hx \,|\, x \in G\},\ (Hx)^g := H(xg);$
  - Left: $H \le G,\ \Omega := \{xH \,|\, x \in G\},\ (xH)^g := (g^{-1}x)H;$
- Special case of the above: Regular actions (with $H = 1$):

Theorem (Cayley)

Every group can be wieved as a subgroup of a symmetric group

- Action by conjugation:
  - On elements: $\Omega := G,\ x^g := g^{-1}xg$
  - On subgroups: $\Omega := \{H \,|\, H \le G\},\ H^g := g^{-1}Hg$

(related concepts: conjugacy classes, centraliser, normaliser)

How to handle permutation groups by computer?

- From now on, $\Omega := \{1, \dots, n\}$, $G \leq S_n$;
- Representing / Storing an element $\in S_n$:
  - An array of length n containing each number $1, \dots, n$ exactly once in some order; (roughly $n \log_2 n$ bits)
  - Cycle decomposition (more difficult to use it in algorithms)

  (Easily convertable to each other)
- Memory requirement: $n \log_2(n)$ bits for a permutation $\in S_n$: This means $4n$ bytes in practice for $n = 10^5$ (we do not care with the 4)
- Current CAS-s can calculate with permutations of degree $n = 10^5$ (even more)
- If we have $2GB$ Memory $\Rightarrow 2GB/4n \approx 5000$ permutations (for $n = 10^5$) can be stored.
- But $|S_{10^5}| = (10^5)! \approx 2.8 \cdot 10^{456574}$;
- How is this possible?

Some ideas

How define a permutation group?

- Example $S_n = \langle (12), (123\,..\,n) \rangle$;
- More generally: Every $G \le S_n$ can be generated by most $n/2$ elements.
- Input group: $X = [x_1, \ldots, x_r] \subset S_n$ with $G = \langle X \rangle$. In practice, usually $|X| \le 10$

How to plan an algorithm?

- Space – Time conflict;
- Store/Calculate elements only when you really need it;
- Avoid long lists;
- Different methods for the same problem – choose the best one (e.g. for degree $n \le 1000$ we store elements to get a faster algorithm, above it we always recalculate them, when we need)

Storing elements

$X \to$ Some algorithm $\to g \in G$ is found. How to handle (store/compute with) g?

- Explicit calculation: g can be written as a product of the generators, so we can calculate it explicitly $\Rightarrow g$ is stored as an array of length n. (It can require both large space and long time)
- Permutation words: g is represented with an array containing pointers to the generators (and their inverses) in the same order as how we should multiply them to get g.
- Straight-line programs (SLP): g is represented with an array $[w_1, \ldots, w_k]$ such that w_i is one of the following for each i:
  - $w_i \in X$;
  - $w_i = (w_j, -1)$ for some $1 \leq j < i$ (take the inverse of w_j);
  - $w_i = (w_j, w_k)$ for some $1 \leq j, k < i$ (take the product of w_j, w_k).
- Storing base images (later)

Example: Calculating and storing elements

Let $X = [a, b]$ and $g = abab^2 \cdots ab^{100}$

- Explicit calculation?
  - Stupid way – Multiply from left to right:
    Time: $2 + \ldots + 101 - 1 = 5149$ multiplication of permutations
    Space: n
  - A bit more clever way

    $d := c := ab;$
    for $i \in [1\,..\,99]$ **do**
    $\quad c := cb;\ d := dc;$

    Time: 199 multiplication of permutations in S_n
    Space: $2n$
- By a permutation word: $g \to [1, 2, 1, 2, 2, 1, 2, 2, 2, 1 \ldots]$.
  Space: 5150 (it does not depend on n)
- By SLP:

$$[a, b, (w_1, w_2), (w_3, w_2), (w_3, w_4), (w_4, w_2), (w_5, w_6), \ldots]$$

Space: 201

Computational Complexity

- big-O notation:
  For $t, f : \mathbb{N} \to \mathbb{R}$ we say $t(n) \in O(f(n))$ if $\exists n_0 \in \mathbb{N},\ c > 0$ s.t. $t(n) < cf(n)$ if $n > n_0$.
- Input length: $O(n)$
- An algorithm is polynomial-time if its running time ($\approx$ the number of steps we need) is in $O(n^c)$ for some $c > 0$ constant.
- Example: Multiplication of two permutations $\in O(n)$.
- Theoretical Computer Science: Fast $\approx$ Polynomial-time
- Practice is often different!
  - Even $O(n^2)$ running-time can be too slow;
  - In some cases, even an exponential-time algorithm can work efficiently in practice.
- Randomisation might help to find solution faster with high probability.

Randomised algorithms

- Deterministic: For the same input you always get the same (correct) output.
- Randomised
  - Monte-Carlo (with error probability $\varepsilon < 1/2$):
    It might give a wrong answer;
    The probability that the answer is wrong is $< \varepsilon$ for every input;
    Reliability can be improved by repeated application.
  - One-sided Monte Carlo: A random algorithm for a decision problem;
    One of the possible answers ('yes' or 'no') is guaranteed to be correct; It can be used as a 'filter'.
  - Las Vegas:
    It never gives an incorrect answer;
    There is a probability $< \varepsilon$ that it does not return an answer at all i.e. reports failure.

Randomised algorithms

Remarks:

- Rerunning Las Vegas algorithm as long as it reports failure $\Rightarrow$ it always give a correct answer, but the running time is random.
- Monte Carlo algorithm $+$ deterministic checking $\Rightarrow$ Las Vegas algorithm.
- In CGT: Random event: Choose a random element from the group.

How to find a random element of a group?

- A group G is given by a set of generators $X = [x_1, \ldots, x_r]$
- Problem: Choose a "random element" of G, i.e. with uniform distribution:
  $\forall g \in G : \; P(g \text{ has been chosen}) = 1/|G|$
- We assume we have a *perfect random generator*, which provides a uniformly random element of a list
- Easy cases:
  - $|G|$ is small enough to list all elements of G;
  - $G = S_n$; (Homework)
  - A base and a strong generating set is known for G; (later)

Homework 1.

Give an algorithm, which provides a uniformly random element of S_n of running time $O(n)$. (with the assumption that you have a perfect random generator, which can choose an element of $[1 \,..\, n]$ in constant time.)

The product replacement algorithm

- Let $X = [x_1, \ldots, x_r]$ be generators for G with $r \geq 10$. Additionally, let $x_0 = 1$.
- Main step:
  - Choose randomly: $s, t \in [1 \mathinner{.\,.} r]$, $s \neq t$, $\varepsilon \in \{\pm 1\}$ and also a "side" from {left,right};
  - Change x_s to either $x_t^{\varepsilon} x_s$ or $x_s x_t^{\varepsilon}$ (depending on which "side" was chosen);
  - Change x_0 to $x_s x_0$ or $x_0 x_s$.
- As an initialistion, run the main step several times. (In practice, 50 step is used)
- After that, each time you need a new random element, run the main step and return with the current value of x_0.

Remarks:

- Fast, usually works well in practice.
- It is not uniformly distributed, and it is unsatisfactory in some cases.

Orbit and stabilizer

Definition (Orbit and stabilizer)

Let G act on Ω.

- The orbit of $\alpha \in \Omega$: $\alpha^G := \{\alpha^g \mid g \in G\}$;
- $\alpha, \beta \in \Omega$ in the same orbit if $\alpha^G = \beta^G$;
- Equivalence classes: Orbits of G on Ω;
- G is transitive: there is just one orbit;
- The stabiliser of $\alpha \in \Omega$ in G: $G_\alpha := \{g \in G \mid \alpha^g = \alpha\} \leq G$.

Theorem (Orbit-stabiliser theorem)

Let $G \leq \Omega$, $\alpha \in \Omega$ and $H = G_\alpha$.

- *There is a bijective correspondence:*

$$\alpha^G \longleftrightarrow \{Hg \mid g \in G\}, \qquad \alpha^g \longleftrightarrow Hg, \ \forall g \in G$$

- $|\alpha^G| = |G : G_\alpha| \Rightarrow |G| = |\alpha^G| \cdot |G_\alpha|$

Basic Orbit algorithm

- The Orbit algorithm:
  - Input: $X = [x_1, \ldots, x_r] \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$, and $\alpha \in \Omega$
  - Problem: Find α^G
  - Maintain an array Δ. At the first step, $\Delta := [\alpha]$.
  - For any $\beta \in \Delta$, calculate β^x for every $x \in X$.
  - Check whether $\beta^x \in \Delta$; If not, we append β^x to Δ.
  - Continue, until $\beta^x \in \Delta$ for every $\beta \in \Delta$, $x \in X$. Then $\Delta = \alpha^G$.
- Membership testing: Use a characteristic vector for $\Delta \subset \Omega$. (This can be a problem if the action is not the natural one)

Pseudocode: The Orbit algorithm

$\textsc{Orbit}(X, \alpha)$

Input: $X \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$, $\alpha \in \Omega$
Output: $\Delta = \alpha^G$

$\Delta := [\alpha]$;
for $\beta \in \Delta$ **do**
    for $x \in X$ **do**
        if $\beta^x \notin \Delta$ **then**
            Append β^x to Δ;
return Δ;

Computing transversals

- Often, we are not only interested in α^G, but for some/every $\beta \in \alpha^G$ also in a $u_\beta \in G$, which moves α to β, i.e. for which $\beta = \alpha^{u_\beta}$.
- $\{u_\beta \mid \beta \in \alpha^G\}$ is a right transversal for G_α.
- Modification of the Orbit algorithm:
  - Maintain an array Δ containing ordered pairs (β, u_β) for $\beta \in \alpha^G$, $\alpha^{u_\beta} = \beta$. Initially, $\Delta = [(\alpha, 1_G)]$;
  - Every time a new element $\gamma = \beta^x$ of α^G if found, (i.e. when there is no element of $(\gamma, *) \in \Delta$) choose $u_\gamma := u_\beta \cdot x$ and append (γ, u_γ) to Δ;
  - At the end of the algorithm, we get an array Δ containing $\{(\beta, u_\beta) \mid \beta \in \alpha^G\}$.

Pseudocode: The Orbit-Transversal algorithm

$\textsc{Orbit-Trans}(X, \alpha)$

Input: $X \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$, $\alpha \in \Omega$

Output: $\Delta = \{(\beta, u_\beta) \mid \beta \in \alpha^G,\ \alpha^{u_\beta} = \beta\}$

$\Delta := [(\alpha, 1_G)]$;
for $(\beta, u_\beta) \in \Delta$ **do**
    for $x \in X$ **do**
        if $(\beta^x, *) \notin \Delta$ **then**
            Append $(\beta^x, u_\beta \cdot x)$ to Δ;
return Δ;

Schreier vectors

- Storing a set of transversals $\{u_\beta \mid \beta \in \alpha^G\}$ requires place $|\alpha^G| \cdot n$. This is n^2 if G is transitive.
- We run out of memory if n is large.
- Solution: Schreier vector. We modify the Orbit algorithm as follows.
  - Besides Δ, we maintain an array Sv indexed by elements $\Omega = \{1, 2, \dots, n\}$.
  - Initalise Sv as $Sv[\alpha] = -1,\ Sv[\beta] = 0$ for $\beta \neq \alpha$.
  - When a new element $\beta^{x_i} \notin \Delta$ found, we not only append β^x to Δ, but we also change $Sv[\beta^{x_i}]$ to i;
  - When the algorithm ends we return Δ, Sv (or just Sv)
- At the end, Sv can also be used as a characteristic vector for α^G, since $\beta \in \alpha^G \iff Sv[\beta] \neq 0$.

Pseudocode: Orbit-Sv

ORBIT-SV(X, α)

Input: $X = [x_1, \ldots, x_r] \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$, $\alpha \in \Omega$

Output: Sv for α

for $i = [1 \,..\, n]$ **do** $Sv[i] := 0$;
$\Delta := [\alpha]$; $Sv[\alpha] := -1$;
for $\beta \in \Delta$ **do**
    for $i = [1 \,..\, r]$ **do**
        if $\beta^{x_i} \notin \Delta$ **then**
            Append β^{x_i} to Δ;
            $Sv[\beta^{x_i}] := i$;
return Sv;

Calculating transversal from Schreier vector

Sometimes we need to explicitly calculate an $u_\beta \in G$ which moves α to β. We can do this from Sv for α as follows.

- In general, it is worth precalculate $X^{-1} := [x_1^{-1}, \ldots, x_r^{-1}]$, since we will use them.
- Input: X, X^{-1}, $\beta \in \Omega$, Sv for α
  Problem: Find an $u_\beta \in G$ with $\alpha^{u_\beta} = \beta$ if $\beta \in \alpha^G$
- First, we check whether $Sv[\beta] = 0$; If yes, then $\beta \notin \alpha^G$ and the algorithm terminates; Otherwise, $\beta \in \alpha^G$.
- By using Sv we step back from β on α^G (by applying some x_k^{-1}-s according to the vector Sv until we reach an $\omega \in \Omega$ satisfying $Sv[\omega] = -1$. Then $\omega = \alpha$ and we get u_β by taking the product of all the x_i-s according to the entries of Sv we touched on the way to α.

Pseudocode: U-beta

U-BETA(β, Sv, X, X^{-1})

Input: $\beta \in \Omega$, a Schreier vector Sv for α
and $X = [x_1, \ldots, x_r], X^{-1} \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$
Output: $u_\beta \in G$ with $\alpha^{u_\beta} = \beta$ if $\beta \in \alpha^G$; otherwise false

if $Sv[\beta] = 0$ **then**
    return false;
$\omega := \beta$; $u := 1_G$; $k := Sv[\omega]$;
while $k \neq -1$ **do**
    $u := x_k u$;
    $\omega := \omega^{x_k^{-1}}$;
    $k := Sv[\omega]$;
return u;

Calculating the stabiliser of α

Theorem (Schreier's Lemma)

Let $H \leq G$ be groups, X: a set of generators for G and $T \ni 1$: a right transversal for H in G. For any $g \in G$ let $\overline{g} := t \in T$ if $Hg = Ht$. Then $Y = \{tx(\overline{tx})^{-1} \mid t \in T, x \in X\} \subset H$ generates H.

Proof.

- $Y \subset H$ by definition;
- Let $g \in H$ and write $g = x_1 \cdots x_m$ by a product of generators;
- Define recursively elements $t_j \in T$ and $y_j \in Y$ by $t_1 = 1,\ t_{j+1} = \overline{t_j x_j}$ and $y_j = t_j x_j (\overline{t_j x_j})^{-1}$; Then $t_j x_j = y_j t_{j+1}$ for $1 \leq j \leq m$. So

$$\begin{aligned} g &= (t_1 x_1) x_2 \cdots x_m = y_1 (t_2 x_2) \cdots x_m = y_1 y_2 (t_3 x_3) \cdots x_m \\ &= y_1 y_2 \cdots y_m t_{m+1} = y_1 y_2 \cdots y_m \in \langle Y \rangle \end{aligned}$$

Calculating the stabiliser of α

We use the previous Orbit-Transversal algorithm, but if we get a β^x which is already in Δ, then we append the Schreier generator $u_\beta x (u_{\beta^x})^{-1}$ to Y.

$\textsc{Orbit-Stabiliser}(X, \alpha)$

Input: $X \subset \mathrm{Sym}(\Omega)$ with $\langle X \rangle = G$, $\alpha \in \Omega$
Output: $\Delta = \{(\beta, u_\beta) \mid \beta \in \alpha^G,\ \alpha^{u_\beta} = \beta\}$,
$Y \subset \mathrm{Sym}(\Omega)$ with $\langle Y \rangle = G_\alpha$

$\Delta := [(\alpha, 1_G)]$;
$Y := [\,]$;
for $(\beta, u_\beta) \in \Delta$ **do**
  for $x \in X$ **do**
    if $\beta^x \notin \Delta$ **then**
      Append $(\beta^x, u_\beta \cdot x)$ to Δ;
    else Append $u_\beta x (u_{\beta^x})^{-1}$ to Y;
return Δ, Y;

How to reduce the number of generators?

- If there is a membership test available, one can check a newly constructed Schreier generator whether it is already in the subgroup generated by the current Y and append to Y ony if it is not.
  - It still not provides a minimal set of generators;
  - It requires many element tests;
- We can choose a relatively small random subset of Y and "hope" that it still generates G_α; (its probability is often very high)
- By using random subproducts of the Schreier generators one can find subsets of Y of moderate size which generate G_α with high probability. Here, a random subproduct of $Y = \{y_1, \ldots, y_s\}$ is an element of the form

$$y_1^{\varepsilon_1} y_2^{\varepsilon_2} \cdots y_s^{\varepsilon_s}, \quad \varepsilon_1, \ldots, \varepsilon_s \in \{0, 1\}.$$

Bases and strong generating sets (BSGS)

Definition

Let $G \leq \mathrm{Sym}(\Omega)$ be a permutation group acting on Ω.

- A sequence $B = (\beta_1, \beta_2, \ldots, \beta_k) \subset \Omega$ is a base for G if $\cap_{i=1}^{k} G_{\beta_i} = 1$;
- The stabiliser chain defined by the base $B = (\beta_1, \ldots, \beta_k)$ is

$$G = G^{(0)} \geq G^{(1)} \geq \ldots \geq G^{(k)} = 1,$$

  where $G^{(i)} := G_{\beta_i}^{(i-1)} = G_{(\beta_1,\ldots,\beta_i)}$ is the subgroup $\{g \in G \,|\, g(\beta_j) = \beta_j, \ \forall\, 1 \leq j \leq i\}$;
- A set of generators $S \subset G$ is a strong generating set for G relative to B if $S \cap G^{(i)}$ generates $G^{(i)}$ for every $0 \leq i \leq k$;
- If $B = (\beta_1, \beta_2, \ldots, \beta_k) \subset \Omega$ is a base for G, then the i-th fundamental orbit Δ_i is the orbit of β_i under the action of $G^{(i-1)}$, i.e. $\Delta_i := \beta_i^{G^{(i-1)}}$.

The importance of BSGS

- Almost every advanced permutation group algorithm uses them;
- Storing group elements with base images:
  If $B = (\beta_1, \beta_2, \ldots, \beta_k)$ is a base for G, then every $g \in G$ is determined by $(\beta_1^g, \beta_2^g, \ldots, \beta_k^g)$
  Most interesting permutation groups in practice has a base of size $\leq 10 \Rightarrow$ very efficient way to store group elements;
- Calculating the order of the group:

$$|G| = |G^{(0)} : G^{(1)}| \cdots |G^{(k-1)} : G^{(k)}| = |\Delta_1| \cdot |\Delta_2| \cdots |\Delta_k|.$$

  By using the orbit algorithm for each pair $(\beta_i, S \cap G^{(i-1)})$ we can calculate each fundamental orbit Δ_i.
- A (perfectly) random element $g \in G$ can be chosen;
- Provides membership test (by shifting);

Finding a random element

- Let U_i be a (right) transversal for $G^{(i)}$ in $G^{(i-1)}$ for every $1 \leq i \leq k$;
  (Such transversals can be find with the orbit-transversal algorithm for input $(\beta_i, S \cap G^{(i-1)})$;)
- We have $G = U_k \times \ldots \times U_1$, i.e. every $g \in G$ can be written of the form $g = u_k \cdots u_1$ in a unique way!
- Generating random element $g \in G$:
  - For every $1 \leq i \leq k$, calculate the i-th fundamental orbit Δ_i;
  - Choose random elements $\gamma_i \in \Delta_i$;
  - Calculate elements $u_i := u_{\gamma_i}$ such that $\beta_i^{u_i} = \gamma_i$;
  - By taking the product $u_k \cdots u_1$, we get a random element of G.

Membership testing (By shifting)

Idea: For a given $g \in \mathrm{Sym}(\Omega)$, we search for a decomposition $g = u_k \cdots u_1$ (with $u_i \in U_i$); such u_i-s can be find $\iff g \in G$.

The shifting algorithm

- Check whether $\beta_1^g \in \Delta_1$: If not, $g \notin G$;
- Otherwise, find $u_1 \in G^{(1)}$ s.t. $\beta_1^g = \beta_1^{u_1}$;
- Continue with gu_1^{-1} and $\beta_2 \ldots$;
- The algorithm terminate in step $m < k$ if $gu_1^{-1} \cdots u_{m-1}^{-1}$ moves β_m outside of Δ_m. In that case, $g \notin G$;
- If you reach the k-th step, then $gu_1^{-1} \cdots u_k^{-1}$ fixes each element of B.

  Then $g \in G \iff gu_1^{-1} \cdots u_k^{-1} = 1$. (Check this!)

Note: Until the last step, you should not explicitly multiply the permutations!

Pseudocode: Shifting

$\textsc{Shifting}(g, B, S, \Delta_*)$

Input: $g \in \mathrm{Sym}(\Omega)$, B, S BSGS,
and Δ_* (reference to the orbit-transversal algorithm)

Output: $m \le k+1$ (the step when terminated),
$h = gu_1^{-1} \cdots u_{m-1}^{-1}$

$h := g$;
for $m \in [1\,..\,k]$ **do**
  $\gamma := \beta_m^h$;
  if $\gamma \notin \Delta_m$ **then**
    return m, h;
  else $h := hu_\gamma^{-1}$;
Return $k+1, h$

Note: $g \in G \iff$ the output is $k+1, 1_G$.

The Schreier–Sims algorithm

Problem: $G \leq \mathrm{Sym}(\Omega)$ is given as $G = \langle X \rangle$. Find a BSGS (B, S).

- Initial step: $B := [\,]$. Extend B to $B := [\beta_1, \dots, \beta_k]$ such that no element of X fixes B pointwise;
- $\forall 1 \leq i \leq k$ let $S^{(i)} := X \cap G_{\beta_1,\dots,\beta_i}$ and $H^{(i)} := \langle S^{(i)} \rangle$. Then

$$G = H^{(0)} \geq H^{(1)} \geq \dots \geq H^{(k)} = 1.$$

Lemma

(B, S) is a BSGS for G $\iff$ $H^{(k)} = 1$ and $H^{(i-1)}_{\beta_i} = H^{(i)}$ for all i.

- For $i = k, k-1, \dots$, we check whether $H^{(i-1)}_{\beta_i} \leq H^{(i)}$. If this holds for each i, then B, S is a BSGS by Lemma;
- Let us assume that this holds for every $l > i$. To check whether $H^{(i-1)}_{\beta_i} \leq H^{(i)}$ for i we take the Schreier generators for $H^{(i-1)}_{\beta_i}$ in $H^{(i-1)}$, and test whether they are in $H^{(i)}$; (Remark: $(\beta_{i+1}, \dots, \beta_k)$ and $S^{(i)}$ is a BSGS for $H^{(i)}$ by assumption, so we can do this!)

The Schreier–Sims algorithm

- If not, we found a $g \in H_{\beta_i}^{(i-1)}$ with $g \notin H^{(i)}$.
- In fact, when we checked $g \notin H^{(i)}$ with algorithm Shifting, it provided us m, h with $i+1 \leq m \leq k+1$ and h such that h fixes $\beta_1, \ldots, \beta_{m-1}$ and
  1. either $m \leq k$ and $\beta_m^h \notin \Delta_m$;
  2. or $m = k+1$ and $h \neq 1$ fixes every element of B.
- In both case, we add h to each of $S^{(i)}, \ldots, S^{(m-1)}$. (Hence we redefine the subgroups $H^{(i)}, \ldots, H^{(m-1)}$ and the fundamental orbits $\Delta_i, \ldots, \Delta_{m-1}$)
- In the second case, we also add a new element β_{k+1} to B not fixed by h, and define $S^{(k+1)} = [\,],\ k := k+1$;
- We start again to check the assumption of Lemma ...
- The algorithm must terminate after finitely many steps; Then (B, S) is a BSGS for G, where $S := \cup_{i=1}^{k} S_i$.

Complexity of Schreier–Sims

The time and space need for calculating a BSGS for $G = \langle X \rangle \leq S_n$ with this (deterministic) algorithm:

- By calculating the transversals explicitly:

  Time: $O(n^2 \log^3 |G| + |X| n^2 \log |G|)$
  Space: $O(n^2 \log |G| + |X| n)$

- By using Schreier vectors:

  Time: $O(n^3 \log^3 |G| + |X| n^3 \log |G|)$
  Space: $O(n \log^2 |G| + |X| n)$

A usual situation is when B is small, and n is large. Then

- Definitely use Schreier vectors;
- Modify algorithms to work with permutation words or SLP-s
- Slowest part: When "Shifting" returns with $k+1, h$, you need to check whether $h = 1_G$.
- "Known-base version" $\Rightarrow$ Fast computation of SGS.

Homework 2.

Prove that if we apply the Shifting algorithm for a group $G = \langle S \rangle \leq S_n$ such that (B, S) is not a BSGS, then it behaves similar to a one-sided Monte Carlo algorithm, except that the error-probability is $\varepsilon > 1/2$. More precisely,

1. If $g \in S_n$ is any permutation such that $g \notin G$, then it still always recognises this fact;
2. On the other hand; if $g \in G$ is chosen with uniform distribution, then the probability that the shifting procedure gives an incorrect answer is at least $1/2$.

Remark: With the help of this, one can define a "Random Schreier–Sims method" which runs much more quickly, and finds a BSGS with prescribed high probability.