Randomisation

Max Neunhöffer

University of St Andrews

GAC 2010, Allahabad

Randomisation

Let $\mathbb{R}$ be the real numbers and $\mathbb{R}^{\geq 0} := \{x \in \mathbb{R} \mid x \geq 0\}$.

Definition (Probability distribution)

Let $\mathcal{E}$ be a finite set. A probability distribution is a map $\mathbb{P} : \mathcal{E} \to \mathbb{R}^{\geq 0}$ with

$$\sum_{E \in \mathcal{E}} \mathbb{P}(E) = 1.$$

We call the elements of $\mathcal{E}$ events and $\mathbb{P}(E)$ the probability of the event E.

Randomisation

Let $\mathbb{R}$ be the real numbers and $\mathbb{R}^{\geq 0} := \{x \in \mathbb{R} \mid x \geq 0\}$.

Definition (Probability distribution)

Let $\mathcal{E}$ be a finite set. A probability distribution is a map $\mathbb{P} : \mathcal{E} \to \mathbb{R}^{\geq 0}$ with

$$\sum_{E \in \mathcal{E}} \mathbb{P}(E) = 1.$$

We call the elements of $\mathcal{E}$ events and $\mathbb{P}(E)$ the probability of the event E.

If $\mathbb{P}(E)$ is the same for all $E \in \mathcal{E}$ then we call $\mathbb{P}$ uniformly distributed.

Randomisation

Let $\mathbb{R}$ be the real numbers and $\mathbb{R}^{\geq 0} := \{x \in \mathbb{R} \mid x \geq 0\}$.

Definition (Probability distribution)

Let $\mathcal{E}$ be a finite set. A probability distribution is a map $\mathbb{P} : \mathcal{E} \to \mathbb{R}^{\geq 0}$ with

$$\sum_{E \in \mathcal{E}} \mathbb{P}(E) = 1.$$

We call the elements of $\mathcal{E}$ events and $\mathbb{P}(E)$ the probability of the event E.

If $\mathbb{P}(E)$ is the same for all $E \in \mathcal{E}$ then we call $\mathbb{P}$ uniformly distributed.

We think of an experiment in which exactly one of the events can happen and $\mathbb{P}(E)$ says, how likely it is that E happens.

Randomisation

Let $\mathbb{R}$ be the real numbers and $\mathbb{R}^{\geq 0} := \{x \in \mathbb{R} \mid x \geq 0\}$.

Definition (Probability distribution)

Let $\mathcal{E}$ be a finite set. A probability distribution is a map $\mathbb{P} : \mathcal{E} \to \mathbb{R}^{\geq 0}$ with

$$\sum_{E \in \mathcal{E}} \mathbb{P}(E) = 1.$$

We call the elements of $\mathcal{E}$ events and $\mathbb{P}(E)$ the probability of the event E.

If $\mathbb{P}(E)$ is the same for all $E \in \mathcal{E}$ then we call $\mathbb{P}$ uniformly distributed.

We think of an experiment in which exactly one of the events can happen and $\mathbb{P}(E)$ says, how likely it is that E happens.

Question

Is random behaviour possible at all?

GAP examples

see other window

Complexity of algorithms

To measure the efficiency of an algorithm, we consider a class $\mathcal{P}$ of problems, that the algorithm can solve.

Complexity of algorithms

To measure the efficiency of an algorithm, we consider a class $\mathcal{P}$ of problems, that the algorithm can solve.

We assign to each $P \in \mathcal{P}$ its size $g(P)$,

Complexity of algorithms

To measure the efficiency of an algorithm, we consider a class $\mathcal{P}$ of problems, that the algorithm can solve.

We assign to each $P \in \mathcal{P}$ its size $g(P)$,

and prove an upper bound for the runtime $L(P)$ of the algorithm for P:

$$L(P) \leq f(g(P))$$

for some function f.

Complexity of algorithms

To measure the efficiency of an algorithm, we consider a class $\mathcal{P}$ of problems, that the algorithm can solve.

We assign to each $P \in \mathcal{P}$ its size $g(P)$,

and prove an upper bound for the runtime $L(P)$ of the algorithm for P:

$$L(P) \leq f(g(P))$$

for some function f.

The growth rate of f measures the complexity.

Complexity of algorithms

To measure the efficiency of an algorithm, we consider a class $\mathcal{P}$ of problems, that the algorithm can solve.

We assign to each $P \in \mathcal{P}$ its size $g(P)$,

and prove an upper bound for the runtime $L(P)$ of the algorithm for P:

$$L(P) \leq f(g(P))$$

for some function f.

The growth rate of f measures the complexity.

Definition

Let $f : \mathbb{R}^+ \to \mathbb{R}^+$. We say that an algorithm has complexity $O(f)$ if there are constants $C, D \in \mathbb{R}^+$ such that its runtime is bounded from above by $C \cdot f(x)$ for all $x \geq D$, where x is the problem size.

Randomised algorithms

Randomised algorithms

Definition (Monte Carlo algorithms)

A Monte Carlo algorithm with error probability ϵ is an algorithm, that is guaranteed to terminate after a finite time, such that the probability that it returns a wrong result is at most ϵ.

Randomised algorithms

Definition (Monte Carlo algorithms)

A Monte Carlo algorithm with error probability ϵ is an algorithm, that is guaranteed to terminate after a finite time, such that the probability that it returns a wrong result is at most ϵ.

Definition (Las Vegas algorithm)

A Las Vegas algorithm with error probability ϵ is an algorithm, that is guaranteed to terminate after a finite time, such that the probability that it fails is at most ϵ.

How to create random numbers?

One usually is content with pseudo-random sequences of numbers.

How to create random numbers?

One usually is content with pseudo-random sequences of numbers.

Example

Let $a, b, m \in \mathbb{N}$ and choose $X_0 \in \{0, 1, \ldots, m-1\}$ arbitrarily. Define then inductively

$$X_i := (a \cdot X_{i-1} + b) \bmod m \in \{0, 1, \ldots, m-1\}.$$

Call X_0 the seed and treat the sequence X_i as random numbers in the range $\{0, 1, \ldots, m-1\}$.

This is not a very good method, but easy to explain.

How to create random numbers?

One usually is content with pseudo-random sequences of numbers.

Example

Let $a, b, m \in \mathbb{N}$ and choose $X_0 \in \{0, 1, \ldots, m-1\}$ arbitrarily. Define then inductively

$$X_i := (a \cdot X_{i-1} + b) \bmod m \in \{0, 1, \ldots, m-1\}.$$

Call X_0 the seed and treat the sequence X_i as random numbers in the range $\{0, 1, \ldots, m-1\}$.

This is not a very good method, but easy to explain.

A good method is the so-called Mersenne-Twister by Matsumoto and Nishimura with a period of $2^{19937} - 1$ and good distribution properties. This is used in GAP.

How to create random numbers?

One usually is content with pseudo-random sequences of numbers.

Example

Let $a, b, m \in \mathbb{N}$ and choose $X_0 \in \{0, 1, \dots, m-1\}$ arbitrarily. Define then inductively

$$X_i := (a \cdot X_{i-1} + b) \bmod m \in \{0, 1, \dots, m-1\}.$$

Call X_0 the seed and treat the sequence X_i as random numbers in the range $\{0, 1, \dots, m-1\}$.

This is not a very good method, but easy to explain.

A good method is the so-called Mersenne-Twister by Matsumoto and Nishimura with a period of $2^{19937} - 1$ and good distribution properties. This is used in GAP.

We assume that we can produce uniformly distributed random numbers!

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$.

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{T_i \mid 1 \leq i \leq k\}$.

Randomised stabiliser computation

Let $\langle T \rangle = G \le \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \le \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{ T_i \mid 1 \le i \le k \}$.
2. Initialise $L := [\,]$ (empty list of generators for S).

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{T_i \mid 1 \leq i \leq k\}$.
2. Initialise $L := [\,]$ (empty list of generators for S).
3. For i in $\{1, 2, \ldots, N\}$ do:

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{T_i \mid 1 \leq i \leq k\}$.
2. Initialise $L := [\,]$ (empty list of generators for S).
3. For i in $\{1, 2, \ldots, N\}$ do:
4. Take a uniformly distributed random $g \in$

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{ T_i \mid 1 \leq i \leq k \}$.
2. Initialise $L := [\,]$ (empty list of generators for S).
3. For i in $\{1, 2, \ldots, N\}$ do:
4. Take a uniformly distributed random $g \in G$.
5. Find j with $xg = xt_j$, this means $gt_j^{-1} \in S$.

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{T_i \mid 1 \leq i \leq k\}$.
2. Initialise $L := [\,]$ (empty list of generators for S).
3. For i in $\{1, 2, \ldots, N\}$ do:
4. Take a uniformly distributed random $g \in G$.
5. Find j with $xg = xt_j$, this means $gt_j^{-1} \in S$.
6. Append gt_j^{-1} as generator for S to L.

Randomised stabiliser computation

Let $\langle T \rangle = G \leq \Sigma_n$ and $x \in \{1, 2, \ldots, n\}$. We want to compute generators for $S := \mathrm{Stab}_G(x)$, let $k := [G : S]$. We have seen the method using $k \cdot (|T| - 1) + 1$ Schreier generators.

Algorithm: Randomised stabiliser computation

Input: $G = \langle T \rangle \leq \Sigma_n$, some $N \in \mathbb{N}$.

1. Enumerate the orbit xG with a Schreier tree.
   $\Longrightarrow$ This gives us a transversal $\{T_i \mid 1 \leq i \leq k\}$.
2. Initialise $L := [\,]$ (empty list of generators for S).
3. For i in $\{1, 2, \ldots, N\}$ do:
4. Take a uniformly distributed random $g \in G$.
5. Find j with $xg = xt_j$, this means $gt_j^{-1} \in S$.
6. Append gt_j^{-1} as generator for S to L.

Output: The list L.

Why does this work?

Lemma (Uniform distribution)

If $g \in G$ is uniformly distributed, then gt_j^{-1} in the above algorithm is uniformly distributed in S.

Why does this work?

Lemma (Uniform distribution)

If $g \in G$ is uniformly distributed, then gt_j^{-1} in the above algorithm is uniformly distributed in S.

Proof: The map $g \mapsto gt_j^{-1}$ is a bijection between the coset St_j and S. ■

Why does this work?

Lemma (Uniform distribution)

If $g \in G$ is uniformly distributed, then gt_j^{-1} in the above algorithm is uniformly distributed in S.

Proof: The map $g \mapsto gt_j^{-1}$ is a bijection between the coset St_j and S. ∎

To make the algorithm work, we need to know what N ought to be. We need results of the following type:

Why does this work?

Lemma (Uniform distribution)

If $g \in G$ is uniformly distributed, then gt_j^{-1} in the above algorithm is uniformly distributed in S.

Proof: The map $g \mapsto gt_j^{-1}$ is a bijection between the coset St_j and S. ∎

To make the algorithm work, we need to know what N ought to be. We need results of the following type:

Theorem (Holt, Roney-Dougal 2010?)

Let $0 < \delta < 1$ and let t be such that $\zeta(t) \leq 2 - \delta$. Let $G \leq \Sigma_n$. If there is a primitive group $H_k \leq \Sigma_n$ and a chain of normal subgroups $G \lhd H_1 \lhd H_2 \lhd \cdots \lhd H_k$, and

$$N \geq 3 \log n + 2 \log\log n + t + 2,$$

then N independent uniformly distributed random elements of G generate G with probability at least δ.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.
4. $T_i := T_i \cdot T_j^e$ and $A := A \cdot T_i$.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.
4. $T_i := T_i \cdot T_j^e$ and $A := A \cdot T_i$.
5. Return the new A.

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.
4. $T_i := T_i \cdot T_j^e$ and $A := A \cdot T_i$.
5. Return the new A.

(Note that after the change the new list still generates G!)

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.
4. $T_i := T_i \cdot T_j^e$ and $A := A \cdot T_i$.
5. Return the new A.

(Note that after the change the new list still generates G!)

To produce a sequence of random elements in G,

1. first execute this a certain number of times,

Product replacement I

We maintain a list $[T_1, T_2, \ldots, T_k]$ of group elements that together generate G and one additional element A.

One product replacement step

1. Pick a random $i \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick a random $j \in \{1, \ldots, k\} \setminus \{i\}$ distributed unif.
3. Pick a random $e \in \{\pm 1\}$ distributed uniformly.
4. $T_i := T_i \cdot T_j^e$ and $A := A \cdot T_i$.
5. Return the new A.

(Note that after the change the new list still generates G!)

To produce a sequence of random elements in G,

1. first execute this a certain number of times,
2. after that, do one step per random element and use the returned element.

Product replacement II

Theorem

Let G be a finite group. If $[T_1, \ldots, T_k]$ is initialised with an arbitrary generating set of G and A is initialised with any group element in G, then, for $N \to \infty$, the distribution of A after N steps converges to the uniform distribution on G.

Product replacement II

Theorem

Let G be a finite group. If $[T_1, \ldots, T_k]$ is initialised with an arbitrary generating set of G and A is initialised with any group element in G, then, for $N \to \infty$, the distribution of A after N steps converges to the uniform distribution on G.

Attention!

It is unknown, how big N has to be to observe a "reasonably good" uniform distribution.
Adjacent elements in the random sequence are by no means guaranteed to be independent.

Product replacement II

Theorem

Let G be a finite group. If $[T_1, \ldots, T_k]$ is initialised with an arbitrary generating set of G and A is initialised with any group element in G, then, for $N \to \infty$, the distribution of A after N steps converges to the uniform distribution on G.

Attention!

It is unknown, how big N has to be to observe a "reasonably good" uniform distribution.
Adjacent elements in the random sequence are by no means guaranteed to be independent.

However: a miracle

Already from $N = 100$ on the sequence of random elements seems to be very good as a random sequence of independent uniformly distributed elements in G.

An accelerator

We maintain a list $[T_0, \ldots, T_k]$ of group elements that together generate G and one additional element A.

Algorithm: One product replacement step

1. Pick random $i, j \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick random $e, f \in \{\pm 1\}$ distributed uniformly.
3. $T_0 := T_0 \cdot T_j^e$.
4. $T_i := T_i \cdot T_0^f$ and $A := A \cdot T_i$.
5. Return the new A.

(Note that after the change the new list still generates G!)

An accelerator

We maintain a list $[T_0, \ldots, T_k]$ of group elements that together generate G and one additional element A.

Algorithm: One product replacement step

1. Pick random $i, j \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick random $e, f \in \{\pm 1\}$ distributed uniformly.
3. $T_0 := T_0 \cdot T_j^e$.
4. $T_i := T_i \cdot T_0^f$ and $A := A \cdot T_i$.
5. Return the new A.

(Note that after the change the new list still generates G!)

Experimental evidence suggests that this mixes faster.

Multiple accumulators

We maintain a list $[T_0, \ldots, T_k]$ of group elements that together generate G an integer m and ℓ additional elements $A_1, \ldots, A_\ell$. Initialise $m := 1$.

Algorithm: One product replacement step

1. Pick random $i, j \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick random $e, f \in \{\pm 1\}$ distributed uniformly.
3. $T_0 := T_0 \cdot T_j^e$.
4. $T_i := T_i \cdot T_0^f$ and $A_m := A_m \cdot T_i$ and $m := (m + 1) \bmod \ell$.
5. Return the new A_m.

(Note that after the change the new list still generates G!)

Multiple accumulators

We maintain a list $[T_0, \ldots, T_k]$ of group elements that together generate G an integer m and ℓ additional elements $A_1, \ldots, A_\ell$. Initialise $m := 1$.

Algorithm: One product replacement step

1. Pick random $i, j \in \{1, 2, \ldots, k\}$ distributed uniformly.
2. Pick random $e, f \in \{\pm 1\}$ distributed uniformly.
3. $T_0 := T_0 \cdot T_j^e$.
4. $T_i := T_i \cdot T_0^f$ and $A_m := A_m \cdot T_i$ and $m := (m + 1) \bmod \ell$.
5. Return the new A_m.

(Note that after the change the new list still generates G!)

This should produce sequences in which adjacent elements are more independent.

Randomised algorithms in practice

The following problems have good randomised algorithms:

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group
- Estimating the orbit length

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group
- Estimating the orbit length
- Group recognition — various problems (see talk number 7)

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group
- Estimating the orbit length
- Group recognition — various problems (see talk number 7)
- Finding normal subgroups

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group
- Estimating the orbit length
- Group recognition — various problems (see talk number 7)
- Finding normal subgroups
- Estimating the group order modulo a normal subgroup

Randomised algorithms in practice

The following problems have good randomised algorithms:

- Computing the point stabiliser of a group action
- Computing a stabiliser chain for a permutation group
- Recognising the isomorphism type of a simple group
- Estimating the orbit length
- Group recognition — various problems (see talk number 7)
- Finding normal subgroups
- Estimating the group order modulo a normal subgroup
- Computing generators for an involution centraliser

Involution centralisers

How can we compute the centraliser of an involution?

Involution centralisers

How can we compute the centraliser of an involution?

The following method by John Bray does the job:

Algorithm: INVOLUTIONCENTRALISER

Input: $G = \langle g_1, \ldots, g_k \rangle$ and an involution $x \in G$.

initialise *gens* $:= [x]$

repeat

$y :=$ RANDOMELEMENT(G)

$c := x^{-1}y^{-1}xy$ **and** $o :=$ ORDER(c)

Involution centralisers

How can we compute the centraliser of an involution?

The following method by John Bray does the job:

Algorithm: InvolutionCentraliser

Input: $G = \langle g_1, \ldots, g_k \rangle$ and an involution $x \in G$.

initialise *gens* $:= [x]$

repeat

$y := \textsc{RandomElement}(G)$

$c := x^{-1}y^{-1}xy$ **and** $o := \textsc{Order}(c)$

if o is even **then**

append $c^{o/2}$ and $(x^{-1}yxy^{-1})^{o/2}$ to *gens*

else

append $z := y \cdot c^{(o-1)/2}$ to *gens*

Involution centralisers

How can we compute the centraliser of an involution?

The following method by John Bray does the job:

Algorithm: InvolutionCentraliser

Input: $G = \langle g_1, \ldots, g_k \rangle$ and an involution $x \in G$.
initialise *gens* $:= [x]$
repeat
    $y :=$ RandomElement(G)
    $c := x^{-1}y^{-1}xy$ **and** $o :=$ Order(c)
    if o is even **then**
        append $c^{o/2}$ and $(x^{-1}yxy^{-1})^{o/2}$ to *gens*
    else
        append $z := y \cdot c^{(o-1)/2}$ to *gens*
until o was odd often enough or gens long enough
return *gens*

Involution centralisers

How can we compute the centraliser of an involution?

The following method by John Bray does the job:

Algorithm: InvolutionCentraliser

Input: $G = \langle g_1, \ldots, g_k \rangle$ and an involution $x \in G$.

initialise *gens* $:= [x]$

repeat

  $y :=$ RandomElement(G)

  $c := x^{-1}y^{-1}xy$ **and** $o :=$ Order(c)

  if o is even **then**

    append $c^{o/2}$ and $(x^{-1}yxy^{-1})^{o/2}$ to *gens*

  else

    append $z := y \cdot c^{(o-1)/2}$ to *gens*

until o was odd often enough or gens long enough

return *gens*

Note: If $xy = yx$ then $c = 1_G$ and $o = 1$ and $z = y$.

Involution centralisers

How can we compute the centraliser of an involution?

The following method by John Bray does the job:

Algorithm: InvolutionCentraliser

Input: $G = \langle g_1, \ldots, g_k \rangle$ and an involution $x \in G$.
initialise *gens* $:= [x]$
repeat
    $y :=$ RandomElement(G)
    $c := x^{-1}y^{-1}xy$ **and** $o :=$ Order(c)
    if o is even **then**
        append $c^{o/2}$ and $(x^{-1}yxy^{-1})^{o/2}$ to *gens*
    else
        append $z := y \cdot c^{(o-1)/2}$ to *gens*
until o was odd often enough or gens long enough
return *gens*

Note: If $xy = yx$ then $c = 1_G$ and $o = 1$ and $z = y$.

And: If o is odd, then z is uniformly distributed in $C_G(x)$.

The End

Bibliography

- John N. Bray.
  An improved method for generating the centralizer of an involution.
  Arch. Math. (Basel), 74(4):241–245, 2000.
- Frank Celler, Charles R. Leedham-Green, Scott H. Murray, Alice C. Niemeyer, and E. A. O'Brien.
  Generating random elements of a finite group.
  Comm. Algebra, 23(13):4931–4948, 1995.
- C. R. Leedham-Green and Scott H. Murray.
  Variants of product replacement.
  In *Computational and statistical group theory (Las Vegas, NV/Hoboken, NJ, 2001)*, volume 298 of *Contemp. Math.*, pages 97–104. Amer. Math. Soc., Providence, RI, 2002.