Zufallszahlen

Zufallszahlen in AntBrain

Spezifikation, Teil II:

> *Zum Beispiel könnte ein Objekt vom Typ* `Match` *die Spielfelder nach jeweils 1000 Spielrunden speichern; bei einer Anfrage nach den Aktionen in der 1012. Runde könnte es dann mit seiner Berechnung in der 1000. Runde anfangen und müsste nur noch zwölf Runden ausrechnen.*

Bei Neuberechnung müssen die gleichen „Zufallszahlen“ benutzt werden wie beim ersten Mal.

Zufallszahlen

Wichtig in vielen Anwendungen

- Spiele (Würfel seit über 5000 Jahren benutzt)
- Stichprobennahme
- Simulation physikalischer Vorgänge
- Approximation schwer lösbarer Probleme durch wiederholte Zufallsexperimente
  - Primaltests mit großer Richtigkeitswahrscheinlichkeit
  - numerische Integration
  - ...
- Kryptographie (Erzeugung zufälliger Schlüssel)

Erzeugung von Zufallszahlen

Echte Zufallszahlen durch physikalische Prozesse

- Würfeln, Münzen werfen
- radioaktive Zerfallprozesse
- atmosphärisches Rauschen
- Quanteneffekte
- Lavalampen, CCD-Kameras im Dunkeln

langsam, benötigt spezielle Hardware

Tabellen von Zufallszahlen, z.B.
RAND Corporation, A Million Random Digits with 100,000 Normal Deviates, 1951
(Taschenbuch-Neuauflage 2002)

Erzeugung von Zufallszahlen

Pseudozufallszahlen

Deterministischer Rechner fängt mit einigen zufälligen Bits (Seed) an und erzeugt daraus eine Folge zufällig *aussehender* Zahlen.

Folge der Zahlen hängt nur vom Seed ab.

schnell, keine Zusatzhardware nötig, wiederholbare Experimente

Beispiel: `java.util.Random`

- Konstruktor `Random(long seed)`
- Konstruktor `Random()` benutzt als Seed die Zeit in ms seit dem 1.1.1970.
- Methoden der Klasse sind komplett deterministisch
- Random ist serialisierbar: beim Serialisieren wird Seed gespeichert.

Kopieren von `Random`-Objekten

Ein Objekt `random` vom Typ `java.util.Random` kann kopiert werden, da diese Klasse serialisierbar ist.

```
Random copy = null;
ObjectOutputStream o = null;
try {
  ByteArrayOutputStream buf = new ByteArrayOutputStream();
  o = new ObjectOutputStream(buf);
  o.writeObject(random);
  ObjectInputStream i = new ObjectInputStream(
        new ByteArrayInputStream(buf.toByteArray()));
  copy = (Random) i.readObject();
} catch (Exception ex) {
  assert(false);
}
```

Jetzt liefern `random` und `copy` die gleiche Folge von „Zufallszahlen“.

Pseudozufallszahlen

Verschiedene Qualitätsanforderungen (z.B. bzgl. Vorhersagbarkeit) für verschiedene Zwecke:

- Windows Solitaire
- Online Casino
- Kryptographie

Beispiel: Debian SSL-Bug (2005–2008)

- Seed für Pseudozufallsgenerator hing nur von Prozessnummer ($\leq 32768 = 2^{15}$) ab.

$\Rightarrow$ Bei der zufälligen Erzeugung von Schlüsseln konnten nur 2^{15} verschiedene Schlüssel herauskommen (statt z.B. 2^{1024} bei einer typischen Schlüssellänge von 1024 Bit).

$\Rightarrow$ Schlecht erzeugte Zufallszahlen machten SSL-Verschlüsselung und SSH-Zugänge unsicher.

Pseudozufallszahlengeneratoren

Verschiedene Verfahren, die verschiedenen Ansprüchen an Geschwindigkeit und Qualität genügen.

- Lineare Kongruenzen (`java.util.Random`)
  schnell, leicht zu implementieren, nicht gut genug z.B. für Anwendungen in der Kryptographie
- Generatoren auf Basis von Hashfunktionen, z.B. für kryptographische Anwendungen (`java.util.SecureRandom`), langsam
- Blum-Blum-Shub-Generator: wenn eine bestimmte komplexitätstheoretische Vermutung gilt, dann kann kein effizienter Algorithmus diese pseudozufälligen Zahlen von echten Zufallszahlen unterscheiden. Für viele Anwendungen zu langsam.

Pseudozufallszahlen durch lineare Kongruenzen

Einfache Methode zur Erzeugung von Pseudozufallszahlen [Lehmer, 1949]

Parameter $a, c, m \in \mathbb{N}$.

Seed $x_0 \in \mathbb{N}$ mit $0 \leq x_0 < m$.

Definiere Folge von natürlichen Zahlen $x_0, x_1, x_2, \ldots$ durch

$$x_{i+1} = a \cdot x_i + c \bmod m$$

Betrachte $\frac{x_1}{m}, \frac{x_2}{m}, \ldots$ als Folge von Zufallszahlen im Intervall $[0, 1)$.

(Die Anzahl der Nachkommastellen, die sinnvoll benutzt werden können, hängt von m ab.)

Pseudozufallszahlen durch lineare Kongruenzen

Beispiel: `java.util.Random`

$$a = 25214903917 \qquad c = 11 \qquad m = 2^{48}$$

$$x_{i+1} = a \cdot x_i + c \bmod m$$

(Diese Zahlen werden seit den 80iger Jahren im Zufallszahlengenerator von Unix System V benutzt.)

Der i-te Aufruf der Funktion `nextInt()` liefert die 32 führenden Bits der 48-Bit-Zahl x_i zurück.

leicht vereinfachter Quellcode:

```
long seed;
public int nextInt() {
  seed = (a * seed  + c) & ((1L << 48) - 1);
  return (int)(seed >> 16);
}
```

Pseudozufallszahlen durch lineare Kongruenzen

Auswahl der Parameter m, a und c

- möglichst große Periode: Die Folge $x_1, x_2, \ldots$ wird nach höchstens m Schritten periodisch

$$x_{i+1} = a \cdot x_i + c \bmod m$$

- statistische Tests, z.B.
  - χ^2-Test: alle Zahlen kommen mit der erwarteten Häufigkeit vor
  - Spektraltest: betrachte Verteilung von Tupeln $(x_i, x_{i+1}, \ldots, x_{i+t})$
  - viele mehr ...

Generator von Blum, Blum und Shub

Auch Pseudozufallsgeneratoren, die Pseudozufallszahlen von hoher Qualität liefern, sind durch einfache Algorithmen gegeben.

Beispiel: Generator von Blum, Blum und Shub (1982)

Folge von Zufallszahlen $x_0, x_1, \cdots \in \mathbb{N}$ definiert durch

$$x_{i+1} = x_i^2 \bmod N,$$

wobei x_0 und N teilerfremd sind und $N = p \cdot q$ für zwei (große) Primzahlen $p \neq q$ mit $p, q = 3 \mod 4$.

Unter der Annahme, dass es keinen effizienten Algorithmus zur Faktorisierung großer Zahlen gibt, kann kein effizienter Algorithmus diese Zufallszahlen von echten Zufallszahlen unterscheiden.