Prakash Punnoor

Über die Grenzen von System.Random

Motivation

- Blog „Fabulous Adventures In Coding“ von Eric Lippert
  - http://ericlippert.com/2013/05/02/producing-permutations-part-six/

- Beispiel des Kartenmischens stellvertretend für Probleme die Folgen von Zufallszahlen benötigen

Anspruch

- Keine physikalische Simulation des Mischens von Karten
- Als Ergebnis ein gemischtes Kartendeck

Reduzierung

- Nummeriere die 52 Karten durch (0 bis 51)

- Reduziere Problemstellung auf:

  Erzeuge Permutation der Zahlen 0 bis 51.

Permutation erzeugen

- Nicht eigenen Permutationsalgorithmus ausdenken, sondern Bewährtes nehmen
  - Gleichverteilung
  - Laufzeit

- → Fischer-Yates
  - Achtung: Bei Implementierung an Vorgabe halten.

Fisher-Yates

```
void FisherYatesShuffle(int[] numbers, Random rand)
{
    for (var i = 0; i < numbers.Length - 1; ++i) {
        var sourceIndex = rand.Next(i, numbers.Length);

        // Tausche Zahl bei sourceIndex mit der bei i
        var temp = numbers[sourceIndex];
        numbers[sourceIndex] = numbers[i];
        numbers[i] = temp;
    }
}
```

Karten mischen

```
int[] GetShuffledDeck()
{
    var numbers = new int[52];
    for (int i = 0; i < numbers.Length; ++i)
        numbers[i] = i;

    var rand = new Random();
    FisherYatesShuffle(numbers, rand);
    return numbers;
}
```

DEMO

Überraschung

- Wie war es möglich
  - in so kurzer Zeit
  - mit nur den ersten 6 bekannten Zahlen

  die gesamte Folge zu berechnen?

Warum genügen 6 Zahlen?

- System.Random ist ein **Pseudo**zufallszahlengenerator
  - Generiert Zahlen **deterministisch** nur abhängig vom Startzustand
    - Praktisch für „reproduzierbaren“ Zufall (Unittests), nicht so praktisch für „echten“ Zufall

Warum genügen 6 Zahlen?

- Startzustand leitet sich aus einer 32 Bit Zahl (Seed) ab
  - nur 31 Bits werden genutzt
    - MSDN: „Wenn eine negative Zahl angegeben wird, wird der absolute Wert der Zahl verwendet.“

→ D.h. es gibt „nur“ etwa 2 Milliarden mögliche Folgen von Zahlen, die generiert werden können.

Theorie

- Im vorliegenden Beispiel gibt es allerdings 52! (Fakultät von 52) Möglichkeiten Zahlen zu permutieren.

- Windows Taschenrechner: $>2^{225}$ ($> 10^{67}$)

  (52 → n! → ln → / → 2 → ln → =)

- Es fehlen mind. 226 - 31 = 195 Bits im Seed (bzw. dazugehörige Random Instanzen.)

Theorie

- System.Random kann daher nicht alle Permutation von 0, ..., 51 erzeugen:
- Für erste Zahl benötigt man 52 Möglichkeiten,
  dann 51, dann 50...
- (Entropiebetrachtung:) Aus allen möglichen Random-Instanzen streicht man durch jede vorgegebene/gewürfelte Zahl alle bis auf jeden 52., dann alle bis auf jeden 51. etc.

Theorie

- Für 6. Zahl verbleiben etwa
  2^{31} / (52 * 51 * 50 * 49 * 48) = 6,886 Random Instanzen
  - Brauchen eigentlich 47!

- Für 7. Zahl findet sich - wenn überhaupt – i.a. noch maximal eine passende Random Instanz (→ Seed).

Angriff

- (Naiver) Brute-Force zur Bestimmung des Seeds:
  - Für alle möglichen Seed-Werte:
    - Random Generator mit aktuellen Seed initialisieren
    - Permutation erzeugen
    - Vergleichen, ob die erzeugte Permutation mit den vorgegebenen ersten 6 Zahlen übereinstimmt
  - Algorithmus lässt sich einfach parallelisieren.
- Hat man genug Zeit oder Geld (→ Cloud) reicht dieser Ansatz

Optimierungen

- Fisher-Yates: Vergleiche während Generieren der Permutation
- Statt Array neu zu initialisieren, Tauschschritte rückgängig machen.
- Parallelisierung

- Mehr „legale“ Optimierungen im Brute-Force sind nicht wirklich möglich

System.Random optimieren

- Sourcen von System.Random besorgen bzw. disassemblieren und hier optimieren:
  - Kleinkram (keine virt. Methoden, Arrayzugriffe) (Speed-up 1,2)
  - Initialisierung nicht mehr durch Konstruktor sondern durch Methode → Objekt wiederverwendbar (Speed-up 1,5)
  - Implementierung in C++ (Speed-up 1)
  - Wechsel auf Linux mit GCC/G++ (Speed-up 1,8)

System.Random optimieren

- Modulo Operation in Schleifen optimieren:

```
for (int i = 0; i < n; ++i) {
    var x = a[h(i)];
    a[i] = f(x);
}
```

System.Random optimieren

Allgemeiner Fall **h(i)**

h(i) vorberechnen und in einem Array ablegen und verwenden

System.Random optimieren

Spezialfall **h(i) = (offset + i) % n**

Schleife in 2 Teilschleifen aufteilen:

```
for (int i = 0; i < n - offset; ++i) {
    var x = a[offset + i];
    a[i] = f(x);
}

for (int i = n - offset; i < n; ++i) {
    var x = a[offset + i - n];
    a[i] = f(x);
}
```

(Speedup 2,7)

System.Random optimieren

- Weitere Compiler-Optimierung einschalten:
  -funroll-loops -ftree-vectorize
  (Speedup 1,9)

- Vektorisierung per Hand (Datenstrukturen ausrichten)
  Schleifen teilweise zusammenlegen (zweite nutzt Ergebnis von erster)
  einfaches Unrolling

  (Speedup 1,3)

Ergebnis

- Hardware:
  AMD Phenom II X4 840 (3,2 Ghz, 4 Cores)

- Software:
  Linux 3.12
  GCC 4.7.3
  (Flags: -O2 -march=barcelona -funroll-loops -std=c++11 -pthread -std=c++11)

- **2^{31} Seeds in 77 Sekunden**

Alternative

- Determinismus von System.Random nicht das primäre Problem sondern niedrige Entropie

- → MS Cryptograpy API nutzender Zufallszahlengenerator in .NET: System.Security.Cryptography. RNGCryptoServiceProvider

RNGCryptoServiceProvider

- Kryptographisch sicher
- Hinreichend viel Entropie, die auch erneuert wird
  - Pool an Entropiequellen
- http://en.wikipedia.org/wiki/CryptGenRandom

RNGCryptoServiceProvider

- Leider nicht ähnliche API wie System.Random:
- GetBytes(byte[]) füllt Array mit zufälligen Werten
- Aus den Bytes muss man sich selbst Zahlen für den gewünschten Bereich ableiten.

Problem: Projektion

- Problemstellung:
  - Haben Generator der Zahlen 0, ..., k-1 erzeugt,
  - wollen Zahlen 0, …, n-1 haben,
  - suchen also geeignete Projektion p(x).

- Annahme: k > n
  - Ansonsten mehrere Zahlen erzeugen und geeignet „konkatenieren“

Projektion 1

- Versuch 1: p(x) = x % n

- Ist nur brauchbar, wenn k >> n (oder n teilt k), denn sonst sind erzeugten Zahlen nicht gleichverteilt:
  - Beispiel: k = 256, n = 100
    0, ..., 55: 50% höhere Wahrscheinlichkeit als 56, …, 99

    0, …, 255 wird folgendermaßen auf 0, …, 99 abgebildet:
    0, …, 99 → 0, …, 99
    100, …, 199 → 0, …, 99
    200, …, 255 → 0, …, 55 ← Problem

Projektion 2

- Versuch 2: p(x) = x % n für x < b
  mit b := k - k % n, ansonsten neues x erzeugen.

- Worst-Case: n = k DIV 2 + 1 (mit n > 1)
  → amortisiert <2 Versuche um x < b zu erzeugen.

- Beispiel: k = 256, n = 100: Damit b = 256 – 56 = 200

Projektion 3

- Versuch 3: p(x) = (int)(x * factor * n)
  mit factor = 1 / (double)(k-1) + c
  mit c > 0 „maximal“, so dass (int)(factor * (k-1)) < 1

- Nur wenig besser als Versuch 1, d.h. keine Gleichverteilung, aber unterschiedliche Wahrscheinlichkeiten über gesamtes Intervall 0, …, k-1 verteilt.
  - System.Random arbeitet auch so

- Beispiel: k = 256, n = 100
  → p(x) = (int)(x * 100/255.1)

Implementierungsideen

- Random.Next(int maxValue) nachbilden auf Basis RNGCryptoServiceProvider:
  - Je nach Größe von maxValue 1-4 Bytes durch Generator erzeugen und diese Bytes „konkateniert“ als Zahl >= maxValue interpretieren.
  - Mittels ProjektionTyp 2 auf gewünschten Bereich abbilden.

Implementierungsideen

- Kleine Verbesserung:
  - Statt ungenutzte Bits der vom Generator erzeugten Bytes „wegzuwerfen“, diese merken und beim nächsten Aufruf von Next() wiederverwenden.

Fazit

- Für „reproduzierbaren Zufall“ oder etwas „Zufallsartiges“ ist System.Random gut geeigent.
- Für „echten“ Zufall oder Anforderungen, die mehr Entropie des Generators erfordern, ist RNGCryptoServiceProvider eine funktionierende Alternative.
- ToDo @Roland: Randomizer Tool analysieren. :)

Danke!