Seminar

Perlen der theoretischen Informatik

Dozenten: Prof. Johannes Köbler und Olaf Beyersdorff

Lineare Kongruenzgeneratoren und Quicksort

Ausarbeitung zum Vortrag

Mia Viktoria Meyer

12. November 2002

Inhaltsverzeichnis

1. Einleitung

Probabilistische Algorithmen, die Zufallszahlen in ihrer Berechnung gebrauchen, sind oftmals die effizientesten Algorithmen, um Probleme zu lösen. Daher liegt es nahe, in die Implementierung eines probabilistischen Algorithmus einen linearen Kongruenzgenerator zu integrieren, der Folgen von Pseudozufallszahlen erzeugt.
Bei der Verwendung eines linearen Kongruenzgenerators für die Implementierung von Quicksort wird sich jedoch zeigen, dass es gewisse Regeln einzuhalten gilt, um eine erwartete Laufzeit O(n log n) zu erzielen - anstelle einer quadratischen erwarteten Laufzeit.

2. Lineare Kongruenzgeneratoren

Ein Pseudozufallszahlengenerator ist ein deterministischer Algorithmus, der aus der Eingabe einer „kleinen“ Anzahl k von echt zufälligen Bits (seed) eine Ausgabe generiert, die einer „großen“ Anzahl m von Bits entspricht (pseudozufällige Bits). Die Folge der m pseudozufälligen Bits ist eine Zufallszahl – in dem Sinne, dass verschiedene Werte mit einer bestimmten Wahrscheinlichkeit angenommen werden -, aber sie ist keineswegs gleichverteilt über $\{0, 1\}^m$, da höchstens 2^k unterschiedliche Werte mit einer Wahrscheinlichkeit größer 0 auftreten können.
Karloff und Raghavan haben den Effekt eines Pseudozufallszahlengenerators auf Quicksort untersucht, der wie folgt definiert ist:

Definition: Ein *linearer Kongruenzgenerator* hat drei Parameter m, a und c. Mit einem zufällig gewählten Startwert (seed) $X_0 \in Z_m$ wird eine Folge von Pseudozufallszahlen X_0, X_1, ..., X_i, ... erzeugt, wobei $X_i = (a * X_{i-1} + c) \bmod m$ für $i \geq 1$.

Für die Implementierung von Quicksort scheint es am naheliegendsten, einen linearen Kongruenzgenerator zu verwenden mit $m = \Theta(n)$, wobei n gleich der Anzahl der zu sortierenden Elemente ist, und mit a und c so gewählt, dass die Periode des linearen Kongruenzgenerators so dicht wie möglich an m liegt. „Periode“ meint in diesem Fall die Anzahl der vom Generator erzeugten Pseudozufallszahlen, bevor sich die Zahlenfolge wiederholt. Überraschenderweise haben Karloff und Raghavan bewiesen, dass Quicksort mit einem solchen linearen Kongruenzgenerator eine quadratische erwartete Laufzeit hat.

3. Annahmen für die Implementierung von Quicksort

Um die spätere Analyse zu erleichtern, werden folgende Annahmen zur Implementierung von Quicksort getroffen:

(Q1) Sind zwei rekursive Teilprobleme zu lösen, so wird immer das kleinere Teilproblem zuerst gelöst.

(Q2) Die Sortierprozedur ist stabil, d.h. die Reihenfolge der Elemente in einer Hälfte ist nach der Umordnung, die durch das Pivotelement induziert wird, dieselbe wie vorher (siehe Abbildung 1).

(Q3) Für jedes Teilproblem der Größe mindestens eins wird die Sortierprozedur rekursiv aufgerufen, d.h. für ein Teilproblem der Größe L werden genau L Pseudozufallszahlen benötigt.

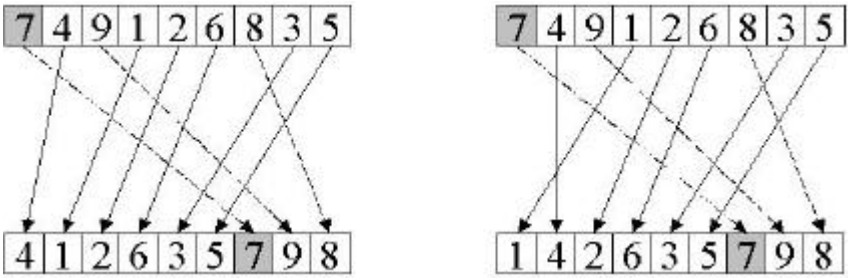

Abbildung 1: Beispiel einer Umordnung durch eine stabile (links) und eine instabile (rechts) Sortierprozedur.

Q1 ist eine beliebte Heuristik, um den Stack zu minimieren, während Q2 und Q3 die Analyse vereinfachen. Obwohl die üblichen in-place Partitionierungsprozeduren nicht stabil sind wie in Q2 gefordert, ist es doch eine wünschenswerte Eigenschaft, auch wenn dabei mehr Vertauschungen notwendig sind. Die Annahme Q3 ist nicht ganz realistisch, da ein Array der Größe eins nicht sortiert werden muss.

Eine Implementierung von Quicksort in Pseudo-Code bei Einhaltung der Annahmen Q1 – Q3 ist wie folgt:

```
procedure Quicksort (var A : array [0 .. n-1])
begin
     if [0 .. n-1] nicht leer then
          i ← nächste Pseudozufallszahl aus dem Bereich [0 .. n-1]
          Wähle das Element p := A[i] als Pivotelement.
          Ordne das Array A – unter Beibehaltung der bisherigen
          Ordnung – so um, dass alle Elemente, die kleiner (größer)
          als das Pivotelement p sind, links (rechts) von p
          angeordnet sind.
          Sei j der Index von p in A nach der Umordnung.
          if [0 .. j-1] kleiner als [j+1 .. n-1] then
               Quicksort (A [0 .. j-1])
```

```
                Quicksort (A [j+1 .. n-1])
            else
                Quicksort (A [j+1 .. n-1])
                Quicksort (A [0 .. j-1])
            end
        end
end
```

4. Annahmen für den linearen Kongruenzgenerator

Der lineare Kongruenzgenerator erfülle folgende Eigenschaften (wobei a, c und m die Parameter seien wie zuvor in der Definition festgelegt und n sei die Anzahl der zu sortierenden Elemente):

(L1) $m > n$.

(L2) $ggT(a, m) = 1$.

(L3) $c = 0$.

(L4) $\min \{t > 0 \mid a^t \equiv 1 \pmod m\} > n/4$.

Durch diese Annahmen wird erreicht, dass die Periode des Generators gleich ¿(n) ist. Dies ist nötig, da nach Annahme Q3 genau n Pseudozufallszahlen erforderlich sind. Deswegen sei auch m > n, da die Periode höchstens die Länge m haben kann. Laut Knuth ist dies nur der Fall, wenn ggT (a, m) = ggT (c, m) = 1 gilt. Zur Vereinfachung der Analyse sei c = 0, so dass für eine maximale Periode nach wie vor ggT (a, m) = 1 berücksichtigt werden muss. Karloff und Raghavan zufolge verläuft die Analyse für ggT (c, m) = 1 ähnlich.
Annahme L4 sagt aus, dass die Periode des Generators eine angemessen große Länge besitzt. Das Minimum t kann als die Periode interpretiert werden, falls der Startwert X_0 gleich 1 ist. Für jeden anderen Startwert X_0 mit ggT (X_0, m) = 1ergibt sich dieselbe Periode, denn die Folge der Pseudozufallszahlen ist dann

X_0, $aX_0 \bmod m$, $a^2X_0 \bmod m$, $a^3X_0 \bmod m$, ..., $a^{t-1}X_0 \bmod m$, $a^tX_0 \bmod m = X_0$,

und existierten zwei verschiedene Folgeglieder mit $a^iX_0 \equiv a^jX_0 \pmod m$ für 0 i < j < t, so würde das implizieren, dass $a^i (a^{j-i} – 1) X_0 \equiv 0 \pmod m$, woraus wiederum folgen würde, dass $a^{j-i} \equiv 1 \pmod m$ (da ggT (X_0, m) = ggT (a, m) = 1), was jedoch der Minimalität von t widerspräche.
Die Annahmen L1 – L4 sind durchaus zugleich erfüllbar. Sei m eine Primzahl, die größer als n ist (L1). In diesem Fall erreichen ö (m-1) Werte von a eine Periode von m-1 für alle m-1 Startwerte X_0 0, wobei ö die Eulersche Funktion ist. Die Anzahl von geeigneten Werten für

a ist somit groß genug, so dass bei einer zufälligen Wahl von a die Chancen für die Erfüllung der Annahmen gut stehen.

5. Verwendung von Pseudozufallszahlen in Quicksort

Eine Methode, damit Pseudozufallszahlen aus Z_m Verwendung finden können in Quicksort bei der Wahl eines beliebigen Pivotelements für ein Teilproblem, dass nur L Werte enthält, lautet wie folgt:

(H1) Sei hash (y, L) = L * y/m , wobei y eine Pseudozufallszahl aus Z_m ist. Dann ist hash (y, L) eine Pseudozufallszahl aus Z_L (für L m).

Dies ist die von Knuth vorgeschlagene Methode, um Pseudozufallszahlen aus Z_m in Pseudozufallszahlen aus Z_L umzuformen.

6. Theorem von Karloff und Raghavan

Für eine Implementierung von Quicksort, die einen linearen Kongruenzgenerator verwendet, der Q1 – Q3, L1 – L4 und H1 genügt, gibt es eine Eingabepermutation, die eine erwartete Laufzeit von Ù ($n^4/m^2 + n \log n$) hat (gemittelt über alle Startwerte X_0).

6.1. Folgerungen

Für m = O (n) ist die erwartete Laufzeit Ù (n^2), was der schlechtesten Laufzeit von Quicksort entspricht. (Im besten Fall hat Quicksort die Laufzeit Ù (n log n).)

6.2. Beweisidee

Es ist leicht, eine Eingabe zu finden, so dass es einen Startwert X_0 gibt derart, dass Quicksort eine erwartete Laufzeit von Ù (n^2) hat. Die Eingabewerte seien so angeordnet, dass die durch den Startwert X_0 festgelegte Folge von Pivotelementen sortiert ist (siehe Abbildung 2). Dies würde jedoch nur zu einer erwarteten Laufzeit Ù ($n^2/m + n \log n$) führen, weil über alle Startwerte X_0 gemittelt wird. Der folgende Beweis wird jedoch zeigen, dass es eine Eingabe gibt, für die $n^2/16m$ Startwerte jeweils ein gleiches Teilproblem der Größe n/4 erzeugen – und zwar genau dann, wenn der lineare Kongruenzgenerator den

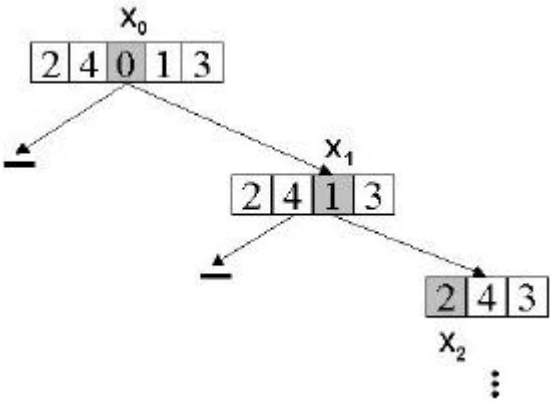

Abbildung 2: Sortierte Folge von Pivotelementen führt zu einer erwarteten Laufzeit Ù (n^2) von Quicksort (worst case).

Wert 1 ausgeben wird. Aufgrund der beschriebenen Struktur hat dieses Teilproblem dann die Laufzeit Ù (n^2) mit der unteren Schranke

$$(n^2/16m * Ù (n^2) + (m - n^2/16m) * Ù (n \log n)) / m = Ù (n^4/m^2 + n \log n),$$

unter Einbeziehung von L1 und der Tatsache, dass Ù (n log n) die bestmögliche Laufzeit für Quicksort für eine beliebige Folge von Pivotelementen ist.

6.3. Beweis

Es wird zwischen zwei Fällen bezüglich m unterschieden:

1. Fall: $m > n^2 / 32$.

Dann gilt: $n^4 / m^2 = O(1)$ und die beste Laufzeit, die Quicksort für eine beliebige Menge von Pivotelementen erzielen kann, ist $Ù (n \log n) = Ù (n^4 / m^2 + n \log n)$.

2. Fall: m $n^2 / 32$.

Sei A [0 .. n-1] das Eingabefeld für Quicksort. A sei als Permutation von {0, 1, ..., n-1} konstruiert, für die $n^2 / 16m$ verschiedene Startwerte X_0 in einer Laufzeit Ù (n^2) resultieren. Sei x so gewählt, dass gelte: $a^{[n/4]}$ x 1 (mod m). x existiert, da gemäß L2 ggT (a, m) = 1 gilt, was zusichert, dass a – und somit auch jede Potenz von a – invertierbar modulo m ist. Darüber hinaus ist zu erwähnen, dass x auch invertierbar modulo m ist (mit der Inverse gleich $a^{[n/4]}$), so dass ggT (x, m) = 1 gilt.

Sei $y_i = a^i x \bmod m$ für alle 0 i n/4 . Die y_i sind paarweise verschieden, denn ggT (x, m) = 1 und folglich gilt die Analyse aus Abschnitt 4.

Sei k = $n^2 / (8m)$ - 1. Sei K = {hash (y_0, n), hash (y_1, n), ..., hash ($y_{n/4}$, n)} \ {0}. |K| kann kleiner als n/4 sein, da verschiedene Werte von y_i durch die Funktion hash auf denselben Wert abgebildet werden können. Durch die Funktion hash ist aber garantiert, dass höchstens

m/n verschiedene Werte aus Z_m auf denselben Wert in Z_n abgebildet werden. Daher gilt also

$$\begin{aligned} |K| &\quad (\quad n/4 \quad - 1) / (\quad m/n \quad) - 1 \\ &> \quad (n / 4) / (m + n)/n - 1 \\ &= \quad n^2 / (4(m + n)) - 1 \\ &\quad n^2 / 8m - 1 \\ &\quad k. \end{aligned}$$

Also gibt es k Werte $y_{i1}, y_{i2}, ..., y_{ik} \in \{y_0, y_1, ..., y_{\quad n/4 \quad - 1}\}$, für die die Werte hash (y_i, n) paarweise verschieden und ungleich null sind.
Die Eingabefolge A werde wie folgt konstruiert:

1. Sei A [hash (y_{ij}, n)] = n – (n/4 - i_j) für 1 j k.
2. Sei ó eine Permutation von {0, 1, ..., n/4 - 1}, für die Quicksort die Laufzeit Ù (n^2) hat, wenn die Folge der Pseudozufallszahlen a, a^2 mod m, a^3 mod m, ... $a^{n/4}$ mod m ist. Sei A [0] = n/4 . Die Werte ó(0), ó(1), …, ó(n/4 - 1) werden in der Reihenfolge den n/4 kleinsten verfügbaren Zellen von A zugeordnet.

Alle Werte in dem Array sind verschieden, da die Werte in 1. sind größer oder gleich 3n/4 und die Werte in 2. sind kleiner oder gleich n/4. Zudem sind alle Werte im Array A in verschiedenen Zellen gespeichert. Bisher sind höchstens k + n/4 + 1 n/8 + n/4 Zellen des Array belegt. Die übrigen Werte werden beliebig zu einer Permutation von {0, 1, ..., n – 1} ergänzt. (Dieser Nichtdeterminismus impliziert, dass Quicksort nicht nur für eine Eingabe, sondern für 5n/8 ! verschiedene Eingaben diese schlechte erwartete Laufzeit hat.)
Was passiert also, wenn $X_0 = y_{ij}$ für ein beliebiges j mit 1 j k. Quicksort sortiert dann A zunächst nach A [hash (y_{ij}, n)] = n – (n/4 - i_j) um, wobei eine Pseudozufallszahl verbraucht wird. Dieser Wert ist größer oder gleich 3n/4, d.h. das kleinere Teilproblem ist die Sortierung der Werte größer als n – (n/4 - i_j). Q1 fordert, das kleinere Teilproblem zuerst zu lösen und Q3 legt fest, dass dabei genau so viele Pseudozufallszahlen verbraucht werden wie Elemente vorhanden sind – also (n – 1) – (n – (n/4 - i_j)) = -1 + n/4 - i_j. Als nächstes wird der lineare Kongruenzgenerator die Pseudozufallszahl

$$\begin{aligned} X_{\quad n/4 \quad - ij} &= y_{ij} \, a^{\quad n/4 \quad - ij} \bmod m \\ &= a^{ij} \, x a^{\quad n/4 \quad - ij} \bmod m \\ &= a^{\quad n/4 \quad} x \bmod m \\ &= 1. \end{aligned}$$

erzeugen. Sei L = n – (n/4 - i_j) die Größe des verbleibenden Teilproblems. Quicksort ordnet dann nach dem Pivotelement

$$A [hash (X_{\quad n/4 \quad - ij}, L)] = \quad A [hash (1, L)]$$

= A [L/m] (H1)

= A [0] (L1)

um.

Laut Q2 hat sich der Inhalt von A [0] = n/4 seit Beginn der Quicksortprozedur nicht verändert. Nach der Umordnung, die durch dieses Pivotelement induziert wird, gilt A [n/4] = n/4 und A [i] = ó(i) für 0 i < n/4 (Q2). Das kleinere Teilproblem wird als nächstes sortiert (Q1) mit einer Laufzeit von Ù (n^2) – wegen der Konstruktion von ó, da die folgenden Pseudozufallszahlen a, a^2 mod m, a^3 mod m, ..., $a^{n/4}$ mod m sein werden.

Für jeden der k = $n^2/(8m)$ - 1 Startwerte y_i erfordert Quicksort eine Laufzeit von Ù (n^2). Daher folgt mit der Annahme des 2. Falles – m $n^2/32$ -, dass k $n^2/(16m)$. Bildet man den Durchschnitt aller m Startwerte, erhält man eine untere Schranke für die erwartete Laufzeit Ù (n^4/m^2 + n log n).

7. Quellen

- **Martin Tompa**: *Lecture Notes on Probabilistic Algorithms and Pseudorandom Generators*, Technical Report #91-07-05, 1991