Weiterbildung "Bayes-Statistik und Simulation", Juni 2012

Das Damenproblem

Es gibt viele Algorithmen, welche den Zufall verwenden, um ein deterministisches Problem zu lösen. Es gibt solche, die trotzdem sicher eine exakte Lösung finden, während andere nur eine approximative Lösung finden, oder mit kleiner Wahrscheinlichkeit eine falsche Lösung liefern. Der vielleicht älteste randomisierte Algorithmus ist das Ziehen einer Stichprobe, um etwas über eine ganze Population zu erfahren. Da man solche Algorithmen ganz auf dem Computer implementieren will, muss man Zufallszahlen auf dem Computer zur Verfügung haben. Meist genügen uniforme Zufallszahlen, Die Laufzeit und die Genauigkeit eines solchen Algorithmus' sind dann im Allgemeinen auch zufällig. Die Verteilung dieser Zufallsgrssen kann man auch mit Simulationen bestimmen.

Das Beispiel, mit dem wir uns hier beschäftigen, wurde von Emo Welzl (D-INFK, ETH) in seiner Einführungsvorlesung gezeigt. Die Aufgabe ist die folgende: Auf einem $n \times n$ Schachbrett sollen n Damen so positioniert werden, dass keine der Damen eine andere bedroht. Mir ist kein Beweis bekannt, dass dieses Problem für jedes $n \geq 5$ eine Lösung hat. In allen Fällen, die ich betrachtet habe, gibt es sehr viele Lösungen. Die Schwierigkeit ist viel mehr, für grosse n eine Lösung schnell zu finden. In jeder Zeile und in jeder Spalte muss genau eine Dame platziert werden, d.h. Lösungen sind Permutationen von $1, 2, \ldots, n$. Offensichtlich ist es aber keine gute Idee, alle Permutationen durchzuprobieren.

In diesem Atelier geht es darum, den folgenden Algorithmus zu verstehen, zu programmieren und den Aufwand von deterministischen und zufälligen Varianten zu vergleichen: Man platziert zuerst eine Dame in der ersten Zeile und dann jeweils eine Dame in der folgenden Zeile, so dass keine Konflikte mit den bereits platzierten Damen entstehen. Ist das nicht möglich, entfernt man so viele Damen, bis man auf einer Zeile ist, wo es noch nicht versuchte Platzierungen gibt. Man wählt dann eine solche Platzierung aus und geht weiter, bis man schliesslich eine Lösung gefunden hat. *Führen Sie diesen Algorithmus von Hand durch für $n = 7$.*

Etwas präziser beschrieben geht der Algorithmus wie folgt: Ein Zustand des Algorithmus ist eine konfliktfreie Platzierung $(\pi_1, \pi_2, \ldots, \pi_j)$ von j Damen in den ersten j Zeilen, sowie Teilmengen $M_1, M_2, \ldots, M_{j+1}$ von $\{1, 2, \ldots, n\}$ von möglichen andern Platzierungen. Das bedeutet $k \in M_i$ falls $k \neq \pi_i$ und $(\pi_i, \ldots, \pi_{i-1}, k)$ eine konfliktfreie Platzierung von i Damen darstellt, die im bisherigen Verlauf des Algorithmus noch nie ausprobiert wurde. Zu Beginn ist $j = 0$ und $M_1 = \{1, 2, \ldots, n\}$. Ein Schritt des Algorithmus verläuft dann wie folgt

1. Setze $j = \max\{i \leq j + 1; M_i \neq \emptyset\}$.
2. Wähle ein $k \in M_j$, entferne dieses k aus M_j und setze $\pi_j = k$.
3. Wenn $j = n$, ist die Lösung gefunden. Sonst berechne die Menge M_{j+1} aller Platzierungen in Zeile $j + 1$, die nicht mit $(\pi_1, \ldots, \pi_j)$ im Konflikt sind.

Verschiedene Versionen des Algorithmus unterscheiden sich dadurch, wie man k in Schritt 2 wählt. Eine deterministische Variante wählt jeweils das kleinste $k \in M_i$. Eine randomisierte Variante wählt k gleichverteilt. *Überzeugen Sie sich, dass dieser Algorithmus korrekt ist und implementieren Sie diesen in der Programmiersprache, mit der Sie am besten vertraut sind.*

Vergleichen Sie die deterministische und die randomisierte Variante für verschiedene n's auf Grund der (mittleren) Anzahl versuchter Platzierungen. Manchmal ist auch der randomiserte Algorithmus langsam. *Überlegen Sie, wie man ihn verbessern könnte.*

R-Funktion zur Lösung des Damenproblems:

```
f.queen <- function(n)
{
  ## Purpose: conflict-free placement of n queens on a n x n chess board
  ## ------------------------------------------------------------------------
  ## Variables:  m = matrix of allowed positions, value = 0
  ##                 if a position is in conflict with placements in
  ##                 previous rows or if we have already tried to place a
  ##                 queen at this position
  ##              p = vector of columns where the queen is placed
  ##              used = indicator whether in one row all possibilities
  ##                     have been tried.
  ## ------------------------------------------------------------------------
  m <- matrix(1,nrow=n,ncol=n)
  j <- 0 # number of queens placed currently
  p <- rep(NA,n)
  npos <- 0 # number of placements made so far
  used <- rep(FALSE,n)
  while (j < n) {
    j <- max((1:(j+1))[used[1:(j+1)]==FALSE])
                 # in which row is the next queen placed?
    ind <- (1:n)[m[j,]==1] # find all allowed columns in row j
    # choosing k = column where new queen is placed
    if (length(ind)==1) {
      k <- ind
      used[j] <- TRUE # no further possibilities in row j
    } else {
        # k <- min(ind) # deterministic choice
        k <- sample(ind, 1) # random choice
      }
    p[j:n] <- c(k,rep(NA,n-j))  #place queen in column k of row j
    m[j,k] <- 0 # position k in row j has now been tried
    npos <- npos + 1 #increase number of placements made so far.
    # Find the columns in row j+1 which are in conflict with previous rows
    if (j<n) {
      m[j+1,] <- 1 # initially no conflicts
      m[j+1,p[1:j]] <- 0 # conflicts along a column
      ind <- p[1:j] + (j:1)
      ind <- ind[ind <= n]
      m[j+1,ind] <- 0 #conflicts along one diagonal
      ind <- p[1:j] - (j:1)
      ind <- ind[ind > 0]
      m[j+1,ind] <- 0 #conflicts along the other diagonals
      used[j+1] <- all(m[j+1,]==0) #check if at least one possibility remains
    }
  }
  list(p,npos)
}
```