Game of Life von Conway

File: conway.rev
in: /home/wiwi/pwolf/lehre/rkurs/programmieren

February 22, 2008

Inhalt

1 Einführung

Was verbirgt sich hinter dem Game of Life?

In diesem Spiel wird eine Population in einer schachbrettartigen Welt betrachtet, die sich über verschiedene Epochen entwickelt. In jeder Zelle des Schachbretts kann ein Individuum leben. Eine Zelle kann aber auch leer sein. Die Veränderung von einer Epoche zur nächsten wird durch drei einfache Regeln beschrieben:

Regel 1: Eine leere Zelle wird in der nächsten Generation besetzt, wenn sie genau drei besetzte Nachbarzellen hat.

Regel 2: Eine besetzte Zelle bleibt auch in der nächsten Generation besetzt, wenn sie zwei oder drei besetzte Nachbarzellen hat.

Regel 3: Alle Zellen, die die Voraussetzungen der Regeln 1 und 2 nicht erfüllen, sind in der nächsten Generation unbesetzt.

2 Die ersten Lösungsschritte bottom-up-mäßig

Zunächst benötigen wir eine Spielwelt, in der unsere Individuen leben. Die leere Welt bilden wir durch eine `(m*n)`-Matrix aus 0-en ab. Individuen repräsentieren wir mittels 1-en. Anfangs weisen wir `k` Zufallszellen eine 1 zu. Wir richten auch gleich den Zähler `epoche` ein, um die Epochen zu zählen

1 ⟨*initialisiere Spiel* 1⟩ ≡
```
leere.welt <- welt <- matrix(0,m,n)
pos.lebend <- sample(1:(n*m),k)
welt[pos.lebend] <- 1
epoche <- 1
```

Wir haben ein paar Variablen verwendet, die wir setzen müssen: die Größe der Welt und den Umfang der Anfangspopulation. Zur Initialisierung des Zufallsgenerators fixieren wir den Zufallsstart.

2 ⟨*start* 2⟩ ≡
```
n <- m <- 20; k <- 80
Random.Start <- 13; set.seed(Random.Start)
⟨initialisier Spiel NA⟩
```

Wie sieht die Welt aus? Eine einfache Populationsanzeige ist schnell gemacht:

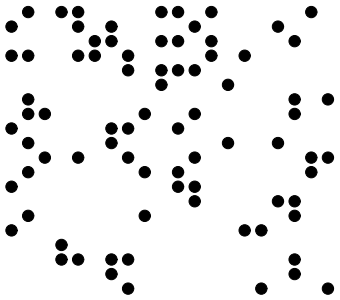
3 ⟨*zeige Welt einfach* 3⟩ ≡
```
o1 <- c(" ","X")[1+welt]; dim(o1)<-dim(welt)
o2 <- cbind("|",o1,"|")
o3 <- rbind("-",o2,"-")
o3[matrix(c(1,1,m+2,m+2,1,n+2,1,n+2),ncol=2)] <- "+"
o4 <- rbind("\n",t(o3)) #; print(o4)
o5 <- paste(o4,collapse=" ")
cat(o5,"\n")
```

So sieht die Anfangswelt aus:

```
+----------------------+
|          x          x|
|        x               |
|    x x  x x            |
|    x                   |
|x                  x x  |
| x         x          x |
|              x     x x |
|x                x x    |
| x         x  x         |
|   x  x    x    x      x|
| x      x         x    x|
|x       x x   x         |
| x x       x   x       x|
| x                     x|
|           x     x      |
|           x  x x       |
|x x   x x  x      x  x   |
|       x x   x x  x     x|
|x     x  x      x      x|
| x  x x      x x  x      |
+----------------------+
```

Schöner ist natürlich eine Darstellung mit Hilfe eines graphischen Fensters. Also plotten wir die Individuen als Punkte!

4 ⟨*zeige Population* 4⟩ ≡
```
x <- col(welt)*welt; x <- x[x!=0]
y <- row(welt)*welt; y <- y[y!=0]
plot(
     x,y,
     xlim=c(0.5,m+0.5),ylim=c(0.5,n+0.5),
     axes=FALSE, bty="n", xlab="",ylab="",
     main=paste(Random.Start,epoche,sep="-"),
     cex=2,pch=16
    )
```



3 Umsetzung der Regeln auch bottom-up-mäßig

Wir wollen die Wirkung der drei Regeln durch die drei Matrizen `R1,R2,R3` beschreiben. Die dritte Regel lässt sich am leichtesten umsetzen:

Regel 3: *Alle Zellen, die die Voraussetzungen der Regeln 1 und 2 nicht erfüllen, sind in der nächsten Generation unbesetzt.*

5 ⟨*ermittle leere Felder nach 3. Regel* 5⟩ ≡
```
if(exists("DEBUG")&&DEBUG) cat("Regel 3")
R3 <- !(R1|R2)
```

Regel 1: Für die erste benötigen wir die Anzahl der Nachbarn. Nehmen wir an, dass es hierfür die Funktion `zaehle.nachbarn()` gibt, die die Anzahl der Nachbarn als (`m*n`)-Matrix ausgibt. Dann folgt `R1` schnell. Zur Erinnerung:

Eine leere Zelle wird in der nächsten Generation besetzt, wenn sie genau drei besetzte Nachbarzellen hat.

6 ⟨*ermittle neue Individuen nach 1. Regel* 6⟩ ≡
```
if(exists("DEBUG")&&DEBUG) cat("Regel 1")
anz.nachbarn<-zaehle.nachbarn()
R1 <- anz.nachbarn==3
```

Regel 2: Die zweite Regel ergibt sich aus der Auswertung des Zustands, der Nachbarschaftszahl und `R1`:

Eine besetzte Zelle bleibt auch in der nächsten Generation besetzt, wenn sie zwei oder drei besetzte Nachbarzellen hat.

7 ⟨*ermittle Überlebende nach 2. Regel* 7⟩ ≡
```
if(exists("DEBUG")&&DEBUG) cat("Regel 2")
# print(dim(welt)); print(dim(R1)); print(dim(anz.nachbarn==2))
R2 <- welt&(R1|anz.nachbarn==2)
```

Damit ist die Vorarbeit durch Entwicklung einiger wahrscheinlich notwendiger Elemente abgeschlossen.

4 Entwurf der Funktion `game.of.life()` – top-down-mäßig

Wir wollen die Funktion, die das Spiel umsetzt, `game.of.life()` nennen. Zweckmäßige Argumente werden sein:

Argument	Bedeutung
`zeilen`	Höhe der Welt
`spalten`	Spalten der Welt
`bevoekerungsgrad`	Anfangsdichte für neues Spiel
`alte.welt`	alte Weltmatrix bei Spielfortsetzung
`zufall`	Zufallsstart für neues Spiel
`DEBUG`	ggf. werden Meldungen ausgegeben

8 ⟨*definiere* `game.of.life` 8⟩ ≡

```
game.of.life <-
function(
         zeilen=20,spalten=20,
         bevoekerungsgrad,
         alte.welt,
         zufall,
         DEBUG=FALSE
        ){
  ⟨checke Argumente 12⟩
  ⟨definiere zaehle.nachbarn 16⟩
  ⟨setze Spiel um 9⟩
  ⟨erstelle Output 11⟩
}
```

Wir wollen uns an die Umsetzung des Spiels wagen:

9 ⟨*setze Spiel um* 9⟩ ≡

```
if(DEBUG) cat("setze Spiel um\n")
for(j in 1:anz.epochen){
  epoche<-epoche+1
  ⟨ermittle neue Individuen nach 1. Regel 6⟩
  ⟨ermittle Überlebende nach 2. Regel 7⟩
  ⟨ermittle leere Felder nach 3. Regel 5⟩
  ⟨ermittle und zeige aktuelle Population 10⟩
}
```

Die ersten drei Chunks haben wir oben schon erstellt. Offensichtlich wird die Wirkung der dritten Regel nicht benötigt. Wenn wir die Überlebenden und die neuen Individuen in eine leere Welt platzieren, benötigen wir übrigens die ermittelte Wirkung der dritten Regel nicht mehr. Das Anzeigen hatten wir auch anfangs gelöst. Damit nicht alles zu schnell geht, lassen wir das System etwas schlafen.

10 ⟨*ermittle und zeige aktuelle Population* 10⟩ ≡

```
if(DEBUG) cat("ermittle und zeige aktuelle Population\n")
welt <- leere.welt
welt[R1|R2] <- 1
⟨zeige Population 4⟩
Sys.sleep(delay)
```

Nach Beendigung geben wir die Population mit ihren Parametern als Liste aus.

11 ⟨*erstelle Output* 11⟩ ≡

```
if(DEBUG) cat("erstelle Output\n")
return(list(welt=welt, epoche=epoche, zufall=Random.Start))
```

Der Check der Argumente kann beliebig weit betrieben werden.

12 ⟨*checke Argumente* 12⟩ ≡
```
if(DEBUG) cat("checke Argumente\n")
if(missing(zeilen)) zeilen<-20
if(missing(spalten)) spalten <- 20
m <- zeilen; n <- spalten

if(missing(alte.welt)){ # neues Spiel
  neues.spiel<-TRUE
  if(missing(bevoekerungsgrad)) bevoekerungsgrad <- 0.5
  k <- floor(m*n*bevoekerungsgrad)
  ⟨initialisiere Spiel 1⟩
  Random.Start <- if(missing(zufall)) 13 else zufall
  anz.epochen <- 10; delay<-0.5
} else {  # Spielfortsetzung
  neues.spiel <- FALSE
  welt <- alte.welt$welt
  m <- nrow(welt); n <- ncol(welt)
  leere.welt <- welt*0
  epoche <- alte.welt$epoche
  Random.Start <- alte.welt$zufall
  anz.epochen <- 100; delay <- 0.1
}
```

5 Test-Aufrufe

Zum Start eines neuen Spiels starten wir:

13 ⟨*starte neues Spiel mit Voreinstellungen* 13⟩ ≡
```
⟨definiere game.of.life 8⟩
alt <- game.of.life()
```

Gerne wollen wir das angefangene Spiel fortsetzen.

14 ⟨*lasse altes Spiel weiterlaufen* 14⟩ ≡
```
⟨definiere game.of.life 8⟩
alt <- game.of.life(alte.welt=alt)
```

Auch soll ein alternativer Start probiert werden.

15 ⟨*starte neues Spiel mit eigenen Setzungen* 15⟩ ≡
```
⟨definiere game.of.life 8⟩
alt <- game.of.life(zeilen=20,spalten=20,
                    bevoekerungsgrad=.5,
                    zufall=7,DEBUG=FALSE)
```

6 Aufgaben

- Die Argument-Liste könnte umfangreicher sein: Wartezeit, Schleifendurchlaufanzahl usw. könnten übergeben werden.
- Ist die Übergabe der Matrix zur Berechnung der Nachbarn nicht ungeschickt? Wie kann man das besser lösen?
- Kann man nicht die jeweils vorherige Generation andeuten und die Geburten durch eine andere Farbe darstellen?

- Gibt es zum Game of Life auch alternative Regeln? Suche welche und setze diese um.

7 Restarbeiten: die Funktion `zaehle.nachbarn()`

Es müssen für alle Punkte der Welt die belegten Nachbarn durchgezählt werden. Das geht am besten, wenn wir die Welt-Matrix um eine Position nach rechts, links, oben und so weiter verschieben und dann einfach die verschobenen Matrizen addieren. Bei Anfügung von aus der Welt herausgeschobenen Elementen auf der Gegenseite ergibt sich quasi eine unendliche Welt.

16 ⟨*definiere* `zaehle.nachbarn` 16⟩ ≡

```
shift.l <- function(welt){ cbind(welt[,-1],welt[,1]) }
shift.r <- function(welt){ cbind(welt[,n],welt[,-n]) }
shift.o <- function(welt){ rbind(welt[-1,],welt[1,]) }
shift.u <- function(welt){ rbind(welt[m,],welt[-m,]) }
shift.lo <- function(welt){  shift.l(shift.o(welt)) }
shift.ru <- function(welt){  shift.r(shift.u(welt)) }
shift.lu <- function(welt){  shift.l(shift.u(welt)) }
shift.ro <- function(welt){  shift.r(shift.o(welt)) }
zaehle.nachbarn<-function(){
  shift.l (welt)+shift.r (welt)+
  shift.o (welt)+shift.u (welt)+
  shift.lo(welt)+shift.ru(welt)+
  shift.lu(welt)+shift.ro(welt)
}
```

Zur Sicherheit soll die Funktion beim Starten definiert werden.

17 ⟨*start* 2⟩+ ≡

```
⟨definiere zaehle.nachbarn 16⟩
```