# Eine Funktion zur Erstellung von Wahrscheinlichkeitsbäumen

File: probtree.rev
in: /home/wiwi/pwolf/lehre/stat1/material

19. Dezember 2011

## Inhalt

# 1 Was ist gewünscht?

Gewünscht ist eine Funktion, die uns aufgrund der Beschreibung einer Folge unabhängiger Zufallsexperimenten einen Wahrscheinlichkeitsbaum generiert. Welche Informationen sind notwendg, um ein solches zusammengesetztes Experiment festzulegen?

- Anzahl der Baumschichten = Anzahl der aufeinander folgenden Zufallsexperimente
- Anzahl der möglichen Ergebnisse des Experimentes für jede Stufe – dabei sei unterstellt, dass die Experimente der einzelnen Stufen voneinander unabhängig sind. Mit dieser Information ist implizit auch die Anzahl der Baumschichten (siehe letzter Punkt) festgelegt.
- Wahrscheinlichkeiten für die Ausfälle der Experimente

Diese Informationen lassen sich gut in einer Liste zusammenstellen, wobei

- jedes Listenelement ein Experiment beschreibt und
- zentral den Vektor der Wahrscheinlichkeiten (evtl. als Text) enthält,
- deren Namen uns die Ergebnisse anzeigen.

## 1.1 Darstellung einfacher Folgen von Zufalls-Experimenten

Für einen dreimaligen Münzwurf ergibt sich als Aufruf:

1 ⟨*test WS-Baum* 1⟩ ≡

```
imax <<- 100; ⟨definiere show.prob.tree NA⟩
show.prob.tree( list( c( "K"=0.5, "Z"=0.5 ),
                      c( "K"=0.5, "Z"=0.5 ),
                      c( "K"=0.5, "Z"=0.5 ) ))
```

Wie können wir eine solche Funktion zusammenbauen?

## 1.2 Umsetzung

Zur Erzeugung eines Wahrscheinlichkeitsbaumes benötigen wir zunächst einen Mechanismus, der eine geeignete Datenstruktur auf Basis der Argumente aufbaut und dann im zweiten Schritt den Baum darstellt. Hiermit können wir folgende Definition notieren.

2 ⟨*start* 2⟩ ≡

```
show.prob.tree <- function(exp.list){
  ⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
  ⟨erstelle interne Repräsentation zu exp.list 18⟩
  ⟨stelle Baum dar 11⟩
  ⟨stelle Baum graphisch dar 22⟩
  ptree
}
```

Jetzt müssen wir nur noch die einzelnen Teilprobleme umsetzen.

# 2 Ein Equipment zur Baum-Repräsentation

## 2.1 Listen-Repräsentation von Bäumen

Bäume sind rekursive Strukturen. Deshalb ist es naheliegend, Bäume durch Listenstrukturen zu repräsentieren. Als erstes benötigen wir eine Funktion zur Generierung eines (fast) leeren Baumes. Den Grenzfall eines knotenlosen Baumes wollen wir ausklammern, so dass die Generierungs-Funktion einen Baum mit einem Knoten erzeugt. Knoten zeichnen sich durch Knoteninformationen aus. Deshalb muss auch für einen Baum mit nur einem Knoten die Information dieses Knotens untergebracht werden. Weiter muss es möglich sein, einem Knoten Kindern anzuhängen.

Diese beiden Qualitäten fassen wir in einer Liste zusammen,

- deren erstes Element die Knoteninformationen aufnimmt und
- deren weitere Elemente jeweils ein Kind des Knotens repräsentieren.

Jedes Kind ist wiederum eine Liste besteht aus Knoteninformation (erstes Element) und Kinder (weitere Elemente).

## 2.2 `new.ptree()`

Mit dieser konzeptionellen Vorgabe können wir die Generierungsfunktion `new.ptree()` aufschreiben.

3 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩ ≡ ⊂ 2, 17, 23, 31, 32

```
new.ptree <- function(top,name){
  if(0==length(names(top)) && !missing(name)) names(top) <- name
  tree <- list( top )
}
```

Ein kleiner Test soll zeigen, dass die Generierungsfunktion wie gewünscht arbeitet.

4 ⟨*erzeuge neuen Baum mit* `new.ptree` 4⟩ ≡ ⊂ 5

```
ptree <- new.ptree("XXX","root")
```

Wir erhalten:

```
[[1]]
 root
"XXX"
```

Die Liste besteht aus nur einem Element, denn die Kinderlosigkeit führt dazu, dass `ptree[-1]` eine leere Liste ist.

5 ⟨*zeige Liste der Kinder von* `ptree` 5⟩ ≡

```
⟨erzeuge neuen Baum mit new.ptree 4⟩
cat("length(ptree)",length(ptree),"/ mode(ptree[-1])",mode(ptree[-1]))
```

Wir erhalten:

```
length(ptree) 1 / mode(ptree[-1]) list
```

## 2.3 `put.el.in.ptree`

Ein Knoten bleibt selten allein. So entwerfen wir als nächstes eine Funktion, die in einen Baum ein weiteres Element einhängt. Dieser Funktion müssen wir Folgendes

mitgeben:

- den Baum, der vergrößert werden soll
- das Element, das eingehängt werden soll
- die Stelle, an der das Element eingehängt werden soll
- evtl. noch ein Attribut für den Knoten.

Die spannendste Geschichte hierbei ist die Angabe des Einfügungsortes. Glücklicherweise kann man in R auf eine verschachtelte Listenstruktur durch Angabe des Pfades, der zu der relevanten Stelle führt, zugreifen. So liefert uns bekanntlich `Liste[[5]]` den Inhalt des fünften Elementes einer Liste, aber mit `Liste[[c(5,3)]]` erhalten wir den Inhalt des dritten Elementes der Liste, die an fünfter Stelle der gesamten Liste zu finden ist. Mit diesem Mechanismus können wir sowohl auf vorhandene Elemente zugreifen, wie auch neue Plätze adressieren.

Wenn wir ein neues Element in einen Baum einfügen, gehen wir davon aus, dass dieses als jüngstes Kind eingesetzt wird. Mit dieser Verabredung besitzen die Kinder eine Reihenfolge, wie wir es etwa von Binärbäumen her kennen. Das zuerst eingefügte Kind bekommt jeweils den Platz zwei zugewiesen, das jüngste Kind wird hinten an die Liste angehängt und `Liste[[5]]` enthält des vierte Kind des Knotens zu dem `"Liste"` gehört. Zur Erinnerung: das erste Element hält die Knoteninformationen. Zwar muss man als Konsequenz bei der Umrechnung von Kinder-Nummern in die Index-Nummern aufpassen, doch wird auf diese Weise eine weitere Verschachtelung mit entsprechenden anderen Fehlermöglichkeiten vermieden.

Mit dieser Vorrede können wir die Funktion `put.el.in.ptree()` definieren.

6 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
put.el.in.ptree<-function(ptree,element,child.no=1,attribut){
  idx.position <-  child.no + 1
  if(!missing(attribut)) names(element) <- as.character(attribut)
  ptree[[ idx.position ]] <- new.ptree(element)
  ptree
}
```

Wir wollen die Einfüge-Funktion sofort testen.

7 ⟨*teste* `put.el.in.ptree()` 7⟩ ≡ ⊂ 10

```
ptree <- new.ptree("XXX","root")
ptree <- put.el.in.ptree(ptree,c("Exp 1-A"),1,"p1")
ptree <- put.el.in.ptree(ptree,c("Exp 1-B"),2,"p2")
ptree <- put.el.in.ptree(ptree,c("Exp 2-L"),c(1,1),"q1")
ptree <- put.el.in.ptree(ptree,c("Exp 2-L"),c(1,2),"q2")
ptree <- put.el.in.ptree(ptree,c("Exp 2-R"),c(2,1),"r1")
ptree <- put.el.in.ptree(ptree,c("Exp 2-R"),c(2,2),"r2")
```

```
[[1]]
 root
"XXX"
[[2]]
[[2]][[1]]
       p1
"Exp 1-A"
[[2]][[2]]
[[2]][[2]][[1]]
```

```
       q1
"Exp 2-L"
[[2]][[3]]
[[2]][[3]][[1]]
       q2
"Exp 2-L"
[[3]]
[[3]][[1]]
       p2
"Exp 1-B"
[[3]][[2]]
[[3]][[2]][[1]]
       r1
"Exp 2-R"
[[3]][[3]]
[[3]][[3]][[1]]
       r2
"Exp 2-R"
```

Wir erkennen, dass unter der Wurzel zwei Kinder zu finden sind, da die Liste der Wurzel drei Elemente enthält: Im ersten finden wir die Knoteninformation `"XXX"`, Im zweiten die Beschreibung von Kind 1 mit Attribut `p1` und Inhalt `"Exp 1-A"` sowie den beiden Enkeln `q1` und `q2`. Das zweite Kind unter der Wurzel, also `ptree[[3]]`, heißt `p2`, hat Inhalt `"Exp 1-B"` und weist uns den Weg zu den Enkeln `r1` und `r2`.

## 2.4 `print.ptree()`

Die oben eingeblendete R-Liste entspricht den Vorstellungen, jedoch ist es für den Betrachter aufgrund der vielen Klammern etwas unübersichtlich. Deshalb entwerfen wir eine kleine Print-Funktion, die nebenbei uns den Umgang mit rekursiven Strukturen zeigt.

Die Grundidee besteht darin, pro Knoten eine Zeile zu drucken und je nach Baumtiefe die auszugebenden Infos nach rechts einzurücken.

Wird `print.ptree()` aufgerufen, wird unter Beachtung der übermittelten Einrücktiefe für die jeweilige Wurzel die Ausgabezeile erstellt und ausgegeben.

Da bei rekursiven Funktion während der Programmierung sich schnell Fehler einschleichen, wurde für die Entwicklungsphase ein Zähler eingebaut, der nach zu vielen Aufrufen einen Abbruch hervorruft.

8 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
print.ptree <- function(x,depth=0,show.names=FALSE){
  ⟨unterbreche Abarbeitung, wenn imax kleiner Null wird 9⟩
  indent <- paste(paste(rep("    ",depth),collapse=""),"+--")
  cat(indent, unlist(x[[1]]),
      if(!is.null(names(x[[1]]))) paste("->",names(x[[1]])))
  children <- x[-1]
  for(i in seq(along=children)){
    print.ptree( children[[i]],depth=depth+1, show.names )
  }
}
```

Der Abbruch-Test für die Entwicklungsphase ist in folgendem Code-Chunk festgelegt.

9 ⟨*unterbreche Abarbeitung, wenn* `imax` *kleiner Null wird* 9⟩ ≡ ⊂ 8, 19, 20

```
if( (imax <<- imax - 1) < 0 ) return("imax == 0 => BREAK!!")
```

Eine Test zur Kontrolle ist empfehlenswert.

10 ⟨*teste* `print.ptree()` 10⟩ ≡
```
⟨teste put.el.in.ptree() 7⟩
imax <<- 100
print.ptree(ptree)
```

Wir erhalten eine wesentlich kompaktere Darstellung der obigen Listenstruktur:

```
+-- XXX -> root
    +-- Exp 1-A -> p1
        +-- Exp 2-L -> q1
        +-- Exp 2-L -> q2
    +-- Exp 1-B -> p2
        +-- Exp 2-R -> r1
        +-- Exp 2-R -> r2
```

Für unser Darstellungsproblem haben wir damit eine Text-Version gefunden.

11 ⟨*stelle Baum dar* 11⟩ ≡ ⊂ 2
```
print.ptree(ptree)
```

# 3 Repräsentation einfacher Wahrscheinlichkeitsbäume

Jetzt haben wir ein Equipment, mit dem wir es wagen, können einfache Wahrscheinlichkeitsbäume auch graphisch zu zeichnen. Das Adjektiv *einfach* hat auch hier die Bedeutung, dass nur Folgen von unabhängigen Zufallsexperimenten betrachtet werden, deren Ausgänge nicht also nicht von vorhergehenden Experimenten abhängen.

Die wesentliche Arbeit besteht darin, die Experiment-Liste durchzugehen und für jedes eine neue Schicht in einem Baum zu organisieren. Diese Formulierung legt einen Ansatz nahe, den Baum von oben nach unten aufzubauen und dieses in einer Schleife über die Einzelexperiment abzuwickeln. Bei einer solchen Vorgehensweise ergibt sich auf jeder Schicht wiederum eine Schleife, die mit zunehmender Schichttiefe umfangreicher wird. Auch hat man hiermit das Problem, viele Baustellen geordnet im Blick zu haben, was schnell Fehler hervorrufen kann. Ein zweiter Ansatz wäre, eine rekursive Umsetzung zu suchen, die wahrscheinlich eine maximale Eleganz besitzen dürfte.

Wir wollen einen Mittelweg gehen und die Aufruf-Intensitäten einer rekursiven Lösung vermeiden und statt dessen eine Job-Menge einführen. Jeder Job soll den Auftrag beschreiben, ein weiteres Element (Experiment) an genau eine Stelle im Bau unterzubringen. Dazu ist die Position der Baustelle als Pfad der wichtigste Teil einer Job-Beschreibung. Denn mit dem Pfad wissen wir, wo angebaut werden muss, andererseits zeigt uns die Pfadlänge an, welches Experiment wir verwenden müssen.

Das Prozedere gestaltet sich sehr übersichtlich: bei jedem Schritt zur Integration eines neuen Knoten muss ein Job aus der Job-Menge entnommen, dann die Funktion `put.el.in.ptree()` aufgerufen und zum Schluss müssen ggf. neue Jobs der Job-Menge hinzugefügt werden.

Diese Arbeit soll die Funktion `exp.list.to.ptree()` erledigen, der die Liste der Experimente mitzugeben ist.

Am Anfang wird ein neuer `ptree` geschaffen und das erste Experiment als Job in die Job-Menge platziert. Dann laufen die beschriebenen Schritte in einer Schleife wiederholt ab, bis die Job-Menge leer ist.

12 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
exp.list.to.ptree <- function(exp.list,top="X",name="ptree"){
  ptree <- new.ptree(top,name=name)
  job.set <- lapply(seq(along=exp.list[[1]]),function(x) x)
  repeat{
    ⟨beende Schleife, wenn Job-Menge leer ist 13⟩
    ⟨hole Job aus Jobmenge 14⟩
    ⟨füge Knoten in Baum ein 15⟩
    ⟨erzeuge ggf. neue Jobs 16⟩
  }
  ptree
}
```

Die Umsetzung der einzelnen Punkte ist zum Teil sehr naheliegend, wie beispielsweise die Schleifenbeendigung.

13 ⟨*beende Schleife, wenn Job-Menge leer ist* 13⟩ ≡ ⊂ 12

```
if(0==length(job.set)) break
```

Die Job-Menge ist eine Liste, so dass wir mit doppelten Indexklammern einen Job extrahieren müssen. Dieser muss der Job-Menge auch explizit entfernt werden.

14 ⟨*hole Job aus Jobmenge* 14⟩ ≡ ⊂ 12

```
job <- job.set[[1]]; job.set <- job.set[-1]
```

Im Zentrum der Einfügung eines Knoten steht der Aufruf von `put.el.in.ptree()`. Diese Funktion muss jedoch mit den passenden Argumenten gefüttert werden. Hierzu wird der Job (der Pfad zur Baustelle) untersucht.

Über die Pfadlänge erhalten wir die Nummer `exp.no` des zu verarbeitenden Experimentes und können leicht damit das Experiment finden. Der letzte Eintrag im Job-Vektor zeigt uns, welchen Experiment-Ausgang wir verarbeiten müssen. Diese beiden Informationen helfen uns den Knoteninhalt (`knot.info`) und die weitere Attribute (`knot.attr`) zu bestimmen.

Damit haben wir die notwendigen Erkenntnisse gesammelt und können losbauen.

- Der Baum `ptree` wird als erstes Element übergeben.
- Der Knoteninhalt wird als einzufügendes Element angesehen.
- Der Job kennzeichnet die Einbaustelle.
- Die Attribute können für ein Knotenattribut verwendet werden.

Es sei auf den Bedeutungswechsel hingewiesen, dass die *Namen von Experiment-Ausgängen* zu Knoteninhalten werden, die Attribute (wie Wahrscheinlichkeiten) zu *Namen von Knoten*. Als Argument mag gelten, dass in anderen Verwendungen neben der Knotenbezeichnung eventuell noch ganz andere Informationen abgelegt werden könnten.

15 ⟨*füge Knoten in Baum ein* 15⟩ ≡ ⊂ 12

```
exp.no <- length(job)
experiment <- exp.list[[exp.no]]
outcome.no <- job[exp.no]
knot.info <- names(experiment)[outcome.no]
knot.name <-       experiment [outcome.no]
```

```
ptree <- put.el.in.ptree(ptree,knot.info,job,knot.name)
```

Zum Schluss müssen wir eventuell noch neue Jobs generieren. Das ist der Fall, wenn die Nummer des gerade verarbeiteten Experimentes kürzer als die Folge der Experimente ist. In diesem Fall müssen wir das nächste Experiment betrachten und ermitteln, wie viele Ergebnisse es hervorbringen kann. Entsprechend dieser Anzahl entstehen neue Jobs bzw. neue Pfade, die sich durch Verlängerung des aktuellen Pfades um die einzelnen Nummern der Versuchsausgänge des neuen Experiments ergeben.

16 ⟨*erzeuge ggf. neue Jobs* 16⟩ ≡ ⊂ 12

```
if( exp.no < length(exp.list) ){
  next.exp <- exp.list[[exp.no+1]]
  new.jobs <- lapply(seq(along=next.exp), function(x)c(job,x))
  job.set  <- c(job.set,new.jobs)
}
```

Ein Test wird hoffentlich zeigen, dass die Einfügungen so verlaufen wie gedacht.

17 ⟨*teste* `exp.list.to.ptree()` 17⟩ ≡

```
imax <<-100
⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
exp.list <- list( Muenzwurf = c( "K"=0.5, "Z"=0.5 ),
                  Dreieck =   c( "eins"=0.3, "zwei"=0.5, "drei"=.2 ),
                  unfair =    c( "oben"=0.4, "unten"=0.6 ) )
ptree <- exp.list.to.ptree(exp.list,top="yyy",name="ptree")
print.ptree(ptree)
```

In der Tat: es klappt!

```
+-- yyy -> ptree
    +-- K -> 0.5
        +-- eins -> 0.3
            +-- oben -> 0.4
            +-- unten -> 0.6
        +-- zwei -> 0.5
            +-- oben -> 0.4
            +-- unten -> 0.6
        +-- drei -> 0.2
            +-- oben -> 0.4
            +-- unten -> 0.6
    +-- Z -> 0.5
        +-- eins -> 0.3
            +-- oben -> 0.4
            +-- unten -> 0.6
        +-- zwei -> 0.5
            +-- oben -> 0.4
            +-- unten -> 0.6
        +-- drei -> 0.2
            +-- oben -> 0.4
            +-- unten -> 0.6
```

Mit dieser Ausrüstung können wir den zweiten Punkt unserer gewünschten Funktion `show.prob.tree()` einlösen.

18 ⟨*erstelle interne Repräsentation zu* `exp.list` 18⟩ ≡ ⊂ 2

```
ptree <- exp.list.to.ptree(exp.list)
```

# 4 Graphische Darstellung

Ausgehend von der Listen-Repräsentation von Wahrscheinlichkeitsbäumen können wir uns jetzt an die graphische Darstellung wagen.

In Abhängigkeit von der Tiefe eines Knotens soll sich die $y$-Koordinate für seine Darstellung ergeben. Die $x$-Koordinate hängt von der Anzahl der möglichen Experiment-Ausgänge der jeweiligen Stufe und den Vorstufen ab.

## 4.1 `find.max.depth()`

Für die Ermittlung des notwendigen Bereiches für $y$ benötigen wir die maximale Tiefe des darzustellenden Baumes. Dazu werden wir die rekursive Funktion `find.max.depth()` verwenden. Diese durchläuft den Baum und gibt der Wert der an jeder Stelle festgestellten maximalen Tiefe zurück.

19 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
find.max.depth <- function(x,depth=0,operation){
  ⟨unterbreche Abarbeitung, wenn imax kleiner Null wird 9⟩
  max.depth <- depth
  for(i in seq(x)){
    if(is.list(x[[i]]))
      max.depth <- max( max.depth, find.max.depth(x[[i]],depth=depth+1) )
  }
  max.depth
}
```

## 4.2 `plot.ptree()`

Die Funktion `plot.ptree()` ist ebenfalls rekursiv. Bei ihrem ersten Aufruf wird ein Koordinatensystem generiert, das dann schrittweise die Baumdarstellung aufnimmt. Hierbei kommt die Funktion `find.max.depth()` zum Einsatz.

Die $x$-Koordinate eines Knotens hängt von der Nummer des jeweiligen Ausgangs sowie von dem Intervall der $x$-Achse ab, das bei den rekursiven Aufrufen jeweils übergeben wird. Zu Beginn wird dieses Intervall auf $[0, 1]$ gesetzt. Jeder Unterbaum erhält ein gleich großes Teilintervall zugewiesen, in dem er sich bzgl. der $x$-Koordinaten ausdehnen darf. Die zentrale Knoteninformation wird mittig in dem zugewiesenen Intervall der $x$-Koordinate eingetragen.

Nach der Knotendarstellung wird die Darstellung der Kinder mittels einer Schleife umgesetzt. Für die einzelnen Durchgänge werden vorweg die Intervalle der Kinder berechnet. Innerhalb eines Schleifendurchgang zeichnen wir die Pfadstücke und fügen ggf. Attribute wie Wahrscheinlichkeiten hinzu. Mit einem rekursiven Aufruf von `plot.ptree()` endet jeder Schleifendurchgang.

20 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
plot.ptree <- function(ptree,x.intervall=0:1,depth=0,srt=0,
                       show.names=FALSE,eps=.05,new=TRUE,topdown=TRUE){
  ⟨unterbreche Abarbeitung, wenn imax kleiner Null wird 9⟩
  if(new){
    max.depth <- find.max.depth(ptree)
    ylim <- c(0,max.depth); if(topdown) ylim <- rev(ylim)
    plot(NULL,xlim=c(0,1),ylim=rev(c(0,3)),bty="n",type="n",
         xlab="",ylab="",axes=FALSE)
  }
```

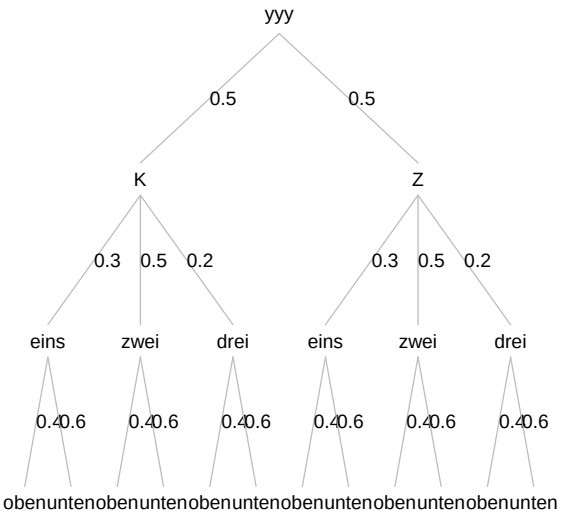
```
  # indent <- paste(paste(rep("    ",depth),collapse=""),"+--")
  # cat(indent, unlist(ptree[[1]]),
  #     if(!is.null(names(ptree[[1]]))) paste("->",names(ptree[[1]])))
  xx <- 0.5*sum(x.intervall); yy <- depth
  text(xx,yy,unlist(ptree[[1]]))
  children <- ptree[-1]
  if( 0 == (n.children <- length(children)) ) return("fertig")
  x.delta <- diff(x.intervall) / n.children
  x.maxs <- x.intervall[1] + x.delta * (1:n.children)
  x.mins <- c(x.intervall[1], x.maxs[-n.children])
  for(i in seq(along=children)){
    x.intervall.i <- c(x.mins[i],x.maxs[i])
    xx.neu <- 0.5*sum(x.intervall.i)
    segments(xx,yy+eps,xx.neu,yy+1-eps,col="gray")
    if(0 < length( txt <- names(children[[i]][[1]]) ))
      text(.5*(xx+xx.neu),.5*(yy+yy+1),txt,adj=1*(xx>xx.neu)*(srt==0),srt=srt)
    plot.ptree( children[[i]],x.intervall.i,depth=depth+1,
                show.names,eps=eps,new=FALSE,srt=srt )
  }
}
```

Diese Funktion muss noch getestet werden.

21 ⟨*teste* `plot.ptree()` 21⟩ ≡

```
imax <<- 100
plot.ptree(ptree,eps=.1)
```

Wir erhalten folgendes Bild.




Damit ist die Darstellung einfacher Wahrscheinlichkeitsbäume abgeschlossen und wird können den obigen Auftrag einlösen:

22 ⟨*stelle Baum graphisch dar* 22⟩ ≡ ⊂ 2

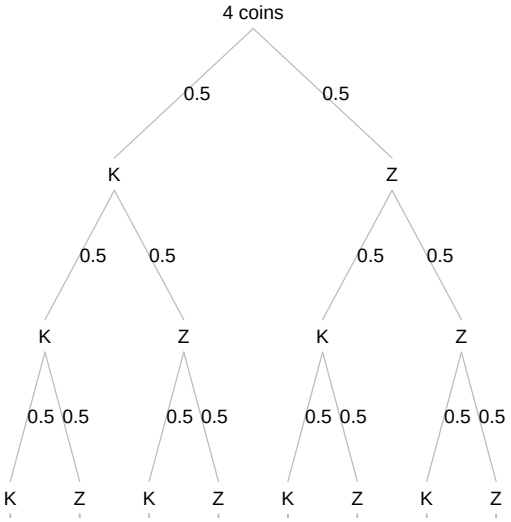
```
plot.ptree(ptree,eps=.1)
```

# 5 Wiederholter Münzwurf

Wie sieht die Darstellung eines 4-maligen Münzwurfes aus?

23 ⟨ * 23⟩ ≡

```
imax <<-100
⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
exp.list <- list( M1 = c( "K"=0.5, "Z"=0.5 ),
                  M2 = c( "K"=0.5, "Z"=0.5 ),
                  M3 = c( "K"=0.5, "Z"=0.5 ),
                  M4 = c( "K"=0.5, "Z"=0.5 ) )
ptree <- exp.list.to.ptree(exp.list,top="4 coins",name="ptree")
plot.ptree(ptree,eps=.1)
```




That's it!

# 6 Abhängige Zufallsexperimente

Was ist, wenn die Zufallsvorgänge nicht unabhängig sind. Stellen wir uns beispielsweise folgende Situation vor: Zuerst ist eine Münze zu werfen; wenn diese *Kopf* zeigt, muss als zweites Experiment ein Würfel, andernfalls noch einmal eine Münze geworfen werden. Vom *Mensch-ärgere-Dich-nicht-Spiel* kennen wir das Warten auf eine Sechs. Auch diese Situation können wir noch nicht darstellen.

**Beschreibungssprache** Zur Lösung müssen wir uns zunächst wieder Gedanken darüber machen, wie wir über solche Folgen abhängiger Experimente sprechen können? Ein Weg besteht darin, erstens eine Liste mit Experimenten wie oben aufzustellen und zweitens eine Adjazenz-Matrix anzugeben, die anzeigt, welche Experimente jeweils folgen. Da solche Adjazenz-Matrixen dünn besetzt sein werden, ist es geschickter, die Nicht-Null-Elemente zu beschreiben. Wir können zum Beispiel für jedes Zufallsexperiment notieren, welches Experiment als nächste durchzuführen ist. Diesem Vorschlag wollen wir folgen, so dass der Anwender zwei Listen angeben muss:

- eine Liste mit Zufalls-Experimenten
- eine gleichlange korrespondierende Liste, die zu jedem Experiment die Nummern aller Folge-Experimente zeigt.

Besitzt beispielsweise das zweite Experiment drei verschiedene Ausgänge, dann muss das zweite Element der Liste mit den Folge-Experiment-Nummern für jeden der drei Ausgänge anzeigen, wie es nach Experiment zwei weitergeht. Also müsste das zweite Element der Folgeliste grundsätzlich ein Vektor der Länge drei sein. Hilfreich können zudem folgende Festlegungen sein: Entsprechend der Replikation von ein-elementigen Vektoren bei Punkt- und Strichrechnung reicht eine Angabe aus, wenn bei allen Ausgängen dasselbe Experiment folgt. Ist mit einem Experiment die Folge der Zufallsexperimente beendet, folgt kein weiteres Element mehr. Dieser Fall soll durch einen `NA`-Eintrag kodiert werden. `NA`-Elemente am Ende der Liste können zur Eingabe-Erleichterung fortgelassen werden.

**Listen-Representation** Damit haben wir einen Vorschlag formuliert, wie ein Anwender einen Wahrscheinlichkeitsbaum und damit das Gefüge von Zufallsexperimenten beschreiben kann. Aus diesen Angaben muss – um den Ideen der bisherigen Auseinandersetzung zu folgen – eine brauchbare Repräsentation in Form einer R-Liste erstellt werden. Hierfür wird im Folgenden die Funktion `exp.seq.to.ptree()` entwickelt, die mit den übrigen schon entworfenen Funktionen kompatibel sein soll.

Die innere Struktur für die neue Funktion übernehmen wir von `exp.list.to.ptree()`. Wieder steuern wir den Listenaufbau über eine Job-Menge. Diese wird initialisiert und in einer Schleife abgearbeitet. Jeder Schleifendurchgang wählt einen Job aus, ermittelt die Attribute eines neuen Knotens, baut diesen ein und erweitert ggf. die Job-Menge. Damit kommen wir zu folgender Struktur:

24 ⟨*stelle Equipment zum Hantieren von Bäumen bereit* 3⟩+ ≡ ⊂ 2, 17, 23, 31, 32

```
exp.seq.to.ptree <- function(exp.list,exp.seq.list,top="X",name="ptree"){
  ⟨initialisiere Baum und Job-Menge für exp.seq.to.tree() 25⟩
  repeat{
    ⟨beende Schleife, wenn Job-Menge von exp.seq.to.tree() leer ist 26⟩
    ⟨hole Job von exp.seq.to.tree() und ermittle Experiment 27⟩
    ⟨finde Knotenbeschreibung, exp.seq.to.tree() 28⟩
    ⟨integriere Knoten in Baum, exp.seq.to.tree() 29⟩
    ⟨erweitere ggf. Job-Menge, exp.seq.to.tree() 30⟩
  }
  ptree
}
```

Die Konstruktion eines leeren Baumes erledigen wir wieder unter Verwendung der Funktion `new.ptree()`. Für die Job-Menge sollten wir noch einmal kurz

zusammenfassen: Ein Job ist der Auftrag einen neuen Kind-Knoten mit seiner Verbindung zu seinem Vorgängerknoten in den Baum einzufügen. Hierbei ist der Vorgängerknoten das lokal betrachtete Zufallsexperiment, zu dem die neue Verbindung einen Ausgang repräsentiert. Für diese Tätigkeit benötigen wir Informationen über die Einfügestelle und zweitens über den Knoten selbst. Zur Klarheit hantieren wir diese beiden Informationstypen getrennt. `job.path.set` wird die Pfade zu den zu erstellenden Knoten sammeln, wogegen `job.exp.set` Informationen über die jeweiligen Experimente hält. Die Pfadmenge konstruieren wir wie bei der Funktion `exp.list.to.ptree()`. Ein Experiment wird durch seine Nummer in der Liste der Experimente beschrieben. Hat das erste Experiment bspw. drei Ausgänge, wird die Pfadmenge mit drei Pfaden gefüllt und die Experiment-Nummer 1 wird dreimal auf dem Vektor `job.exp.set` abgelegt.

25 ⟨*initialisiere Baum und Job-Menge für* `exp.seq.to.tree()` 25⟩ ≡ ⊂ 24

```
ptree <- new.ptree(top,name=name)
job.path.set <- lapply(seq(along=exp.list[[1]]),function(x) x)
job.exp.set  <- rep(1,length(job.path.set))
```

Für den Schleifenabbruch orientieren wir uns ebenfalls an der Liste `job.path.set`. Wenn diese leer ist, ist nichts mehr zu tun.

26 ⟨*beende Schleife, wenn Job-Menge von* `exp.seq.to.tree()` *leer ist* 26⟩ ≡ ⊂ 24

```
if( 0 == length(job.path.set)) break
```

Eine Job aus der Job-Menge zu holen, bedeutet, die Einfügestelle aus `job.path.set` und die zugehörige Experiment-Nummer aus `exp.no` zu entnehmen. Das Experiment bekommen wir per Zugriff auf die `exp.list`.

27 ⟨*hole Job von* `exp.seq.to.tree()` *und ermittle Experiment* 27⟩ ≡ ⊂ 24

```
path   <- job.path.set[[1]]; job.path.set <- job.path.set[-1]
exp.no <- job.exp.set[[1]];  job.exp.set  <- job.exp.set[-1]
experiment <- exp.list[[exp.no]]
```

Welche Informationen gehören zu einen Knoten? Wichtig ist die Nummer des neuen zu zeichnenden Verbindung. Dieses ist gerade die Zahl, die als letzte im Pfad abgelegt ist. Mit dieser `outcome.no` können wir mit Hilfe des Experimentnamens die Information erfahren, die wir in den Knoten schreiben müssen. Weitere Attribute – wie Wahrscheinlichkeiten – sind dem Namen des Experimentes zu entnehmen.

28 ⟨*finde Knotenbeschreibung,* `exp.seq.to.tree()` 28⟩ ≡ ⊂ 24

```
outcome.no <- path[length(path)]
knot.info <- names(experiment)[outcome.no]
knot.attr <-       experiment [outcome.no]
```

Zur Knotenintegration verwenden wir wieder die Funktion `put.el.in.ptree`.

29 ⟨*integriere Knoten in Baum,* `exp.seq.to.tree()` 29⟩ ≡ ⊂ 24

```
ptree <- put.el.in.ptree(ptree,knot.info,path,knot.attr)
```

Als letzte Tätigkeit gilt es noch, die Job-Menge mit neuen Aufträgen zu füllen, sofern noch etwas ergänzt werden muss. Wenn die Liste der Folgeexperimente `exp.seq.list` kürzer ist als die aktuelle Experiment-Nummer anzeigt, gibt es keine Folgeexperimente. Wenn die Nummer des Ausgangs größer ist als die Länge der in `exp.seq.list` abgelegten Folgeexperimente, wird das zuerst genannte Experiment repliziert. Die Nummer des ermittelten Folgeexperimentes wird auf `next.exp` abgelegt. Wenn etwas zu tun ist, ermitteln wir die neuen Pfade durch anhängen der Outcome-Nummern an den alten Pfad und verlängern die Menge `job.path.set`. Zu jedem Job bzw. neu zu zeichnenden Pfadstück muss die

Nummer des verursachenden Experimentes bekannt sein. Deshalb verlängern wir den Vektor `job.exp.set` solcher Experimente um `next.exp` in der Häufigkeit der Anzahl der Outcomes.

30 ⟨*erweitere ggf. Job-Menge,* `exp.seq.to.tree()` 30⟩ ≡ ⊂ 24

```
if( length(exp.seq.list) < exp.no) next
idx <- if( length(exp.seq.list[[exp.no]]) < outcome.no) 1 else outcome.no
next.exp <- exp.seq.list[[exp.no]][idx]
if(!is.na(next.exp)){
  new.paths <- lapply(seq(along=exp.list[[next.exp]]), function(x)c(path,x))
  job.path.set <- c(job.path.set,new.paths)
  new.exps  <- rep(next.exp,length(new.paths))
  job.exp.set  <- c(job.exp.set,new.exps)
}
```

Damit ist die Definition der Funktion beendet und wir können uns einem Test-Chunk widmen.

31 ⟨*teste* `exp.seq.to.tree()` 31⟩ ≡

```
⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
imax <<- 200
exp.list <- list( M1 = c( "E"=0.2, "Z"=0.3, "D"=0.5 ),
                  M2 = c( "EK"=0.5, "EZ"=0.5 ),
                  M3 = c( "ZK"=0.4, "ZZ"=0.6 ),
                  M4 = c( "DK"=0.4, "DZ"=0.7 ),
                  M5 = c( "DKa"=0.4, "DZa"=0.7 ),
                  M6 = c( "VK"=0.2, "VZ"=0.8 ) )
seq.list <- list( 2:4, 5, 6, NA, NA, NA, NA, NA )
# seq.list <- list( 1:3, NA, NA, NA )
tr <- exp.seq.to.ptree(exp.list,seq.list)
print.ptree(tr)
plot.ptree(tr,eps=.1)
```

Im Rahmen des Starts sollten wir gleich eine zentrale Funktion zur allgemeinen Darstellung von Bäumen und das Equipment auch allgemein bereitstellen.

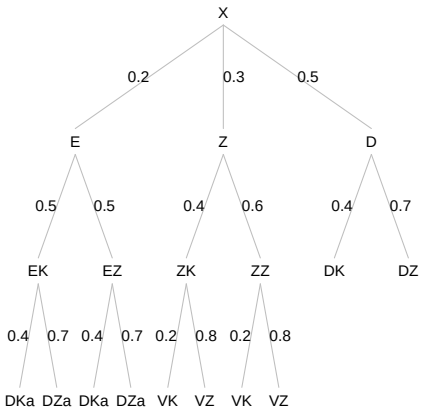
32 ⟨*start* 2⟩+ ≡

```
⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
show.tree <- function(exp.list){
  ⟨stelle Equipment zum Hantieren von Bäumen bereit 3⟩
  tr <- exp.seq.to.ptree(exp.list,seq.list)
  print.ptree(tr)
  plot.ptree(tr,eps=.1)
  tr
}
```

So, mit einem letzten Test beenden wir den Entwicklungsteil.

33 ⟨*Beispiel* 33⟩ ≡

```
exp.list <- list( M1 = c( "E"=0.2, "Z"=0.3, "D"=0.5 ),
                  M2 = c( "EK"=0.5, "EZ"=0.5 ),
                  M3 = c( "ZK"=0.4, "ZZ"=0.6 ),
                  M4 = c( "DK"=0.4, "DZ"=0.7 ),
                  M5 = c( "DKa"=0.4, "DZa"=0.7 ),
                  M6 = c( "VK"=0.2, "VZ"=0.8 ) )
seq.list <- list( 2:4, 5, 6, NA, NA, NA, NA, NA )
imax <<- 200
show.tree(exp.list)
```




# 7 Fazit

In diesem Papier haben wir einen kleinen Werkzeugkasten zur Darstellung von Wahrscheinlichkeitsbäumen entworfen. Die Diskussion hat uns nebenbei gezeigt, wie wir ganz allgemein Bäume mit Listenstrukturen in R repräsentieren und wie wir typische Probleme des Aufbaus und der Traversierung lösen können. Bei solchen Fragestellungen übernehmen häufig rekursiven Algorithmen die Arbeit. Als zweite Möglichkeit wurde vorgeführt, wie man mit Hilfe einer Job-Menge die sonst vom Rekursionsmechanismus zu leistende Verwaltungsarbeit erledigen kann. Auf diese Weise hat man während der Entwicklung auch den Vorteil, leichter Zwischenzustände überprüfen zu können. Natürlich sind damit nicht alle Fragen gelöst, die uns bei Bäumen begegnen. Beispielsweise wäre eine Funktion interessant, die einen schiefen Baum balanziert. Der interessierte Leser kann als Übung einmal überlegen, wie man mit dem Werkzeugkasten einen binären Suchbaum aufbauen und darstellen kann. Als letzte Anregung sei vorgeschlagen, über ein Repräsentation des zu zeichnenden Baumes nachzudenken, bei der man mit den einzelnen darzustellenden Graphik-Objekten leicht hantieren kann, bspw. um interaktiv die Lage von Knoten zu verändern.

# Anhang

### Object Index

## Code Chunk Index

⟨*test WS-Baum* 1⟩ ..... p2
⟨*unterbreche Abarbeitung, wenn* `imax` *kleiner Null wird* 9⟩ ⊂ 8, 19, 20 ..... p5
⟨*xxx* 34 ∪ 35 ∪ 36⟩ ..... p**??**
⟨*zeige Liste der Kinder von* `ptree` 5⟩ ..... p3