Handlungsreisenden-Problem per genetischem Algorithmus

File: genetic-algos.rev
in: /home/wiwi/pwolf/lehre/aud/material

December 6, 2010

Inhalt

1 Einleitung

Das Problem des Handlungsreisenden ist ein Optimierungsproblem, das sich zur Demonstration verschiedener algorithmischer Ansätze eignet. Im Internet findet man längere Ausführungen unter der Adresse

`http://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`

Weiter sei auf den Link

`http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

hingewiesen. In diesem Papier stellen wir das Prinzip genetischer Algorithmen vor und demonstrieren es an dem kurz mit TSP (traveling salesman problem) bezeichneten Problem.

2 Genetische Algorithmen

Ziel genetischer Algorithmen ist es, sehr gute Lösungen zu Aufgabenstellungen zu finden, bei denen ein Auffinden der an sich gewünschten optimalen Lösung beispielsweise aus Gründen kombinatorischer Komplexität misslingt. So gibt es beim Handlungsreisendenproblem mit 10 Orten bereits $10! = 3628800$ Touren bzw., wenn nur Routen mit festem Startpunkt ohne Umlaufsinn interessieren, immerhin noch $9!/2 = 181440$ Lösungen. Ein Ausprobieren aller Touren scheitert deshalb schon bei kleinen Ortsanzahlen und man ist gezwungen, Ideen zur Verkleinerung dieser Menge zu suchen oder sich mit ganz guten Lösungen zufrieden zu geben.

Kern des Ansatzes ist es, eine Menge von Problemlösungen Schritt für Schritt zu verändern. Bei den Veränderungen bleiben sehr gute Lösungen erhalten, sehr schlechte werden mit einer großen Wahrscheinlichkeit ausgesondert und auf Basis von Variationen und Kombinationen entstehen neue Lösungen. Hinter der simultanen Verfolgung mehrerer Lösungen steckt die Erwartung, dass man durch Zusammenführung von Teilen guter Lösungen noch bessere findet und außerdem nicht in einem lokalen Optimum hängen bleibt.

Entstanden ist dieser Ansatz durch Übertragung von Konzepten aus der Biologie, genauer der Genetik in den Bereich der Optimierungsprobleme. Den Ursprung wird der Leser schon an den Begrifflichkeiten erkannt haben. Bei genetischen Algorithmen werden einzelne Lösungen als Individuen und Mengen von Lösungen als Population angesehen. Während der Durchführung des Algorithmus wird von Epoche zu Epoche die Anfangspopulation verändert. Veränderungen erfolgen dabei durch Mutation und Rekombination der Erbmasse. Am Ende soll die Population mindestens eine sehr gute Lösung enthalten. Damit sehr gute Lösungen nicht verloren gehen, müssen deren Überlebenswahrscheinlichkeiten sehr hoch sein im Gegensatz zu relativ schlechten Lösungen.

Damit sind die Schritte des Vorgehens vorgezeichnet:

1. konstruiere Anfangspopulation

2. berechne Qualitäts- oder Fitness-Maß der Individuen
3. falls Abbruchkriterium erfüllt, liefere Ergebnis ab
4. verändere Population: Mutationen, Rekombinationen, Aussterben, ...
5. gehe zum zweiten Schritt zurück

Als größtes Problem muss für die Anwendung des Ansatzes die Problemstellung so transformiert werden, dass die relevanten Größen einer Lösung als Gene, die auf einer Schnur aufgefädelt sind, angesehen werden können. Für die Umsetzung des Evolutionsprozesses sind eine Reihe von Verfahrensparameter festzulegen, wie:

- Wie können sich Individuen verändern? → Welche Mutationen?
- Wie können Individuen Erbinformationen austauschen? → Welche Arten von Crossing-Over?
- Welche Wahrscheinlichkeiten sollen für Überleben, Sterben, Mutation oder Crossing-Over gelten?
- Welche Populationsgröße soll betrachtet werden?
- Wie viele Generationen sollen verfolgt werden oder aber was führt zur Beendigung des Evolutionsprozesses?

Im Folgendem wollen wir die Umsetzung der Ideen anhand des TSP vorführen.

2.1 Problemübertragung

Bevor das Verfahren starten kann, muss ein Weg gefunden werden, eine Lösung durch eine Reihe von Eigenschaften – durch ihre Gene – zu beschreiben. Für das TSP soll die Reihenfolge der besuchten Orte die Erbinformation bilden. Wir können uns die einzelnen Elemente der Folge als einzelne Gene vorstellen. Damit fallen uns schnell verschiedene Möglichkeiten der Mutation ein: Beispielsweise können wir benachbarte Elemente oder ganze Stücke austauchen. Entsprechend können wir auch zusammenhängende Ortsfolgen einer Lösung entnehmen und sie in eine andere einzupflanzen. Dabei müssen wir natürlich darauf achten, die alten Stellen der neu eingefügten Orte zu löschen.

Die Qualität einer Tour durch die zu besuchenden Orte drückt sich in der Weglänge aus. Als Fitness-Maß werden wir den Kehrwert der Tourlänge verwenden, damit ein höherer Wert mit einer besseren Tour einhergeht.

2.2 Ein spezielles TSP-Problem

Damit wir ein konkretes Problem zur Hand haben, genieren wir einige Zufallsorte. Für die Rekonstruierbarkeit setzen wir vorher den Start des Zufallszahlengenerators. Die Anzahl der Orte `n` ist ein Problemparameter, nicht jedoch ein Verfahrensparameter.

1 ⟨*define a special problem* 1⟩ ≡ ⊂ 7
```
n.orte <- 20
set.seed(13); xy <- cbind(rnorm(n.orte),rnorm(n.orte))
```

Später werden wir zur Berechnung von Tourenlängen immer wieder Distanzen benötigen. Deshalb ermitteln wir die Euklidischen Distanzen zwischen den einzelnen Orten.

2 ⟨*compute distance matrix* 2⟩ ≡ ⊂ 7, 18
```
orte.dist <- as.matrix(dist(xy,upper=TRUE))
```

3 Umsetzung einzelner Schritte

3.1 Generierung der Start-Population

Als Vorbereitungsschritt ist eine Startpopulation zu wählen. Wir erzeugen uns Individuen mittels Zufalls-Permutationen der Zahlen von 1 bis `n.pop` und merken uns die Startpopulation auf der Variablen `pop`. `pop` wird später die jeweils aktuelle Population repräsentieren. Mit Hilfe der Variablen `zz.initial` wird der Start des Zufallszahlengenerators gesetzt.

3 ⟨*initialize population* 3⟩ ≡ ⊂ 7, 18
```
set.seed(zz.initial)
pop <- sapply(1:n.pop,create.random.individuals)
```

Technisch erledigen wir die Aufgabe der Generierung von Individuen mit der Funktion `create.random.individual()`. Auf diese Weise erhalten wir einen sprechenden Code und können später schnell Alternativen einbauen. Es folgt die Definition der Funktion.

4 ⟨*define some functions* 4⟩ ≡ ⊂ 7, 18
```
create.random.individuals <- function(x) sample(1:n.orte)
```

3.2 Fixierung einiger Verfahrensparameter

Welche Parameter sind für unseren Algorithmus bedeutsam? Ein offensichtlicher Parameter ist die Populationsgröße, andere wurden schon oben motiviert. Wir fassen die Verfahrensparameter der Übersicht wegen in einer Tabelle zusammen.

Parameter	Bedeutung
`n.pop`	Populationsgröße
`n.generations`	Anzahl der Generationen, die man verfolgen möchte
`n.stay.alive.anyway`	Anzahl der besten, die auf jeden Fall in der nächsten Epoche weiterleben
`n.drop.off`	Anzahl der schlechtesten, die in einer Epoche entfernt werden
`prob.mutation`	Wahrscheinlichkeit für Mutation eines Individuums
`prob.crossing`	Wahrscheinlichkeit für Crossing-Over für ein Individuum
`n.final.plots`	technischer Parameter, der die Anzahl der dargestellten Individuen am Ende festlegt
`n.interim.plots`	technischer Parameter, der die Anzahl von Plots festlegt, die während der Evolution erstellt werden
`n.alive.by.sampling`	Anzahl der Individuen, die mit einer gewissen Chance weiterleben (wird berechnet)
`zz.initial`	Zufallszahlengeneratorstart

5 ⟨*set parameters* 5⟩ ≡ ⊂ 7
```
n.pop               <- 40
n.generations       <- 200
n.stay.alive.anyway <- 5
n.drop.off          <- 5
prob.mutation       <- 0.2
prob.crossing       <- 0.2
n.final.plots       <- 20
n.interim.plots     <- 20
n.alive.by.sampling <- n.pop-n.stay.alive.anyway-n.drop.off
zz.initial          <- 17
```

Neben diesen Werten müssen noch die schon angesprochene Überlebenswahrscheinlichkeiten festgelegt werden, die von der Qualität der einzelnen Lösungen abhängen sollten. Diese *variablen* Größen setzen wir durch eine Funktion um, in der die Wahrscheinlichkeiten proportional zu den Fitwerten berechnet werden.

6 ⟨*define some functions* 4⟩+ ≡ ⊂ 7, 18
```
prob.stay.alive <- function() fits/sum(fits)
```

3.3 Umsetzung des Algorithmus

Mit diesen Überlegungen ist der Anfang geklärt und wir können überlegen, was zur Durchführung einer Epoche zu tun ist. Natürlich müssen wir zu jeder Lösung unser Qualitätsmaß `fits` ausrechnen. Dann stellen wir die nächste Generation aus alten und neuen Individuen zusammen. Auf diese neuen Population wenden

wir die Veränderungsmechanismen zur Mutation und zum Crossing-Over an. Sowohl für die Darstellung als auch für Berechnung ist es hilfreich, die Individuen gemäß ihrer Fits zu sortieren. Dieses ist zwar nicht unbedingt notwendig, jedoch erleichtert es einige Zugriffe. Wir kommen damit zu folgender groben Struktur:

7 ⟨*start genetic algorithm* 7⟩ ≡ ⊂ 16

```
⟨define some functions 4⟩
⟨define a special problem 1⟩
⟨compute distance matrix 2⟩
⟨set parameters 5⟩
⟨initialize population 3⟩
⟨open graphics device 15⟩
for(i.generation in 1:n.generations){
  ⟨find fit and sort members 9⟩
  ⟨show interim results 14⟩
  ⟨find new population 10⟩
  ⟨execute some mutations 11⟩
  ⟨execute some crossing-overs 12⟩
}
⟨find fit and sort members 9⟩
⟨show final results 13⟩
```

3.4 Berechnung der Fitness

Damit wir Individuen vergleichen können, müssen wir deren Qualität berechnen. Das soll die Funktion `fitness` leisten, die zu dem ihr übergebenen Individuum den Kehrwert der Weglänge durch Zugriff auf `orte.dist` ermittelt.

8 ⟨*define some functions* 4⟩+ ≡ ⊂ 7, 18

```
fitness<-function(members){
  fit <- 1/sum(orte.dist[cbind(members,c(members,members[1])[-1])])
  fit
}
```

Mit Hilfe der Funktion `fitness` finden wir die Fitness-Werte aller Individuen und sortieren dann die Population gemäß dieser Indikatoren absteigend. Damit befindet sich die beste Lösung in der ersten Spalte von `pop`.

9 ⟨*find fit and sort members* 9⟩ ≡ ⊂ 7, 17, 18

```
fits   <- apply(pop,2,fitness)
idx    <- order(fits,decreasing=TRUE)
pop    <- pop[,idx]; fits<-fits[idx]
l.ways <- 1/fits
```

3.5 Ermittlung der nächsten Generation

Die nächste Generation finden wir, indem wir die besten `n.stay.alive` Elemente der Population beibehalten und die `n.drop.off` Elemente durch neue ersetzen. Die weiteren Plätze in der neuen Population werden zufällig vergeben,

wobei die Wahrscheinlichkeiten wie beschreiben von den Fitness abhängen. Aufgrund der Sortierung stehen die besten Lösungen in `pop` vorn und die schlechtesten ganz hinten.

10 ⟨*find new population* 10⟩ ≡ ⊂ 7, 17, 18

```
# best members will stay alive anyway
pop.new   <- pop[,1:n.stay.alive.anyway]
# replace members being dropped off by new random ones
pop.new   <- cbind(pop.new,sapply(1:n.drop.off,create.random.individuals))
# sample members of populations to find those that will stay alive
idx.alive <- sample(1:n.pop,n.alive.by.sampling,TRUE,prob=prob.stay.alive())
pop       <- cbind(pop.new,pop[,idx.alive])
```

3.6 Mutationen und Crossing-Over

Für Mutationen fallen uns eine Reihe von Ideen ein. Wir wollen jedoch nur ganz einfache ins Auge fassen. Dazu bestimmen wir mit `rbinom` zunächst die Anzahl der Individuen, die überhaupt mutiert werden sollen. Bei jeder Mutation gehen wir in folgender Weise vor:

1. Wähle per Zufall ein Individuum aus. Hierbei ist zu beachten, dass das Individuum mit dem besten Fit nicht gewählt werden kann.
2. Wähle zwei Gene (Orte) des gewählten Individuums aus.
3. Tausche die Position der Gene (Orte) aus.

Dieser Vorschlag führt zu einer Umsetzung mit einer Schleife.

11 ⟨*execute some mutations* 11⟩ ≡ ⊂ 7, 17, 18

```
# how many mutations
n.mut <- rbinom(1,n.pop,prob.mutation)
# loop along the members to be mutated
for(i in seq(n.mut)){
  # find member to be mutated but not the first one
  k <- sample(2:n.pop,1)
  # find two positions for exchange
  two.pos <- sample(1:n.orte,2)
  # change in member k the two values
  pop[two.pos,k] <- pop[two.pos[2:1],k]
}
```

Der Austausch von Erbinformationsbereichen zwischen zwei Individuen wird durch folgende Prozedur umgesetzt:

1. Wähle zwei Individuen aus.
2. Bestimme eine Schnittstelle, ab der die Information ausgetauscht wird.
3. Sichere für die beiden Individuen jeweils die Stücke hinter der Schnittstelle.

4. Entferne von jedem der beiden Individuen die Gene (Orte), die die fremden Stücke beinhalten.

5. Ergänze die fremden Stücke zur Komplettierung der Erbinformation jedes Individuums.

Diese Prozedur wird für jedes Crossing-Over umgesetzt, wobei die Anzahl der Austausche durch Realisation einer binomialverteilten Zufallsvariablen bestimmt wird. Es sei bemerkt, dass zunächst die fremden Stücke hinten an das verkürzte Erbgut angefügt wurden. Dieses führte dazu, dass die Anfänge immer recht stabil blieben. Inzwischen wird mit verbessertem Effekt das abgeschnittene fremde Stück vorn an die Tour gesetzt.

12 ⟨*execute some crossing-overs* 12⟩ ≡ ⊂ 7, 17, 18

```
# how many crossing-overs
n.crossing<-rbinom(1,n.pop,prob.crossing)
# loop for crossing over
for(i in if(n.crossing>0) 1:n.crossing else NULL){
  # find two members for crossing over
  k <- sample(2:n.pop,2)
  # find position for crossing over
  j <- sample(1:n.orte,1)
  # find tails of individuals
  p1 <- pop[,k[1]];           p2 <- pop[,k[2]]
  tail1 <- p1[j:n.orte];      tail2 <- p2[j:n.orte]
  # remove items that will be replaced
  p1 <- p1[-match(tail2,p1)]; p2 <- p2[-match(tail1,p2)]
  # perform crossing over:
  p1 <- c(tail2,p1);          p2 <- c(tail1,p2)
  # store changes in set of individuals
  pop[,k]<-cbind(p1,p2)
}
```

3.7 Ergebnisdarstellungen

Am meisten interessiert uns beim TSP, in welcher Reihenfolge wir die Orte ansteuern sollen und wie lang die zugehörige Wegstrecke ist. Für die Einschätzung der gefundenen Strecke und für die Einschätzung der Population ist es hilfreich, die Touren der Populations-Elemente zu zeichnen. Dazu öffnen wir ein neues Graphik-Device, wählen gemäß der gewünschten Anzahl von Plots Individuen aus und zeichnen die zugehörigen Touren. Als Legende heften wir den Plots die Indizes der Individuen in der sortierten Population und den Fitness-Wert an.

13 ⟨*show final results* 13⟩ ≡ ⊂ 7, 17, 18

```
cat("beste Tour:",pop[,1])
cat("Strecke:",l.ways[1])
idx<-floor(seq(1,n.pop,length=n.final.plots))
for(i in 1:n.final.plots){
  plot(xy[c(pop[,idx[i]],pop[1,idx[i]]),],type="l",col="red",lwd=3,
       axes=FALSE,xlab="",ylab="")
  points(xy[c(pop[,idx[i]],pop[1,idx[i]]),],lwd=2,col="black")
  points(xy[pop[1,i],1],xy[pop[1,i],2],col="blue",pch=16)
```

```
  text(-1.5,1.5,"Ind:"); text(-1.5,1.0,idx[i])
  text( 1.5,1.5,"Fit:"); text( 1.5,1.0,round(l.ways[idx[i]],1))
  box()
}
print(l.ways)
```

Während der Berechnung kann es hilfreich sein, die Entwicklung zu verfolgen. Dazu wird die jeweils beste Tour gezeichnet, jedoch nicht für jede Generation, sondern nur für jede i-te Generation, wobei die Schrittweite durch `n.generations/n.interim.plots` festgelegt ist.

14 ⟨*show interim results* 14⟩ ≡ ⊂ 7, 17, 18

```
if(!exists("show.graphics") || show.graphics==TRUE){
  if(i.generation==1){
    step<-ceiling(n.generations/n.interim.plots)
    # cat("step",step)
    plot(xy[c(pop[,1],pop[1,1]),],axes=FALSE,xlab="",ylab="")
    lines(xy[c(pop[,1],pop[1,1]),],col="green",lwd=3)
    text(-1.5,1.5,"Gen:"); text(-1.5,1.0,i.generation)
    text( 1.5,1.5,"Fit:"); text( 1.5,1.0,round(l.ways[1],1))
    box()
  }
  if(0==(i.generation%%step)){
    plot(xy[c(pop[,1],pop[1,1]),],axes=FALSE,xlab="",ylab="")
    lines(xy[c(pop[,1],pop[1,1]),],col="green",lwd=3)
    text(-1.5,1.5,"Gen:"); text(-1.5,1.0,i.generation)
    text( 1.5,1.5,"Fit:"); text( 1.5,1.0,round(l.ways[1],1))
    points(xy[pop[1,i],1],xy[pop[1,i],2],col="blue",pch=16)
    box()
  }
}
```

Graphiken erfordern ein Output-Device. Falls ein solches verlangt ist, wird ein neues geöffnet.

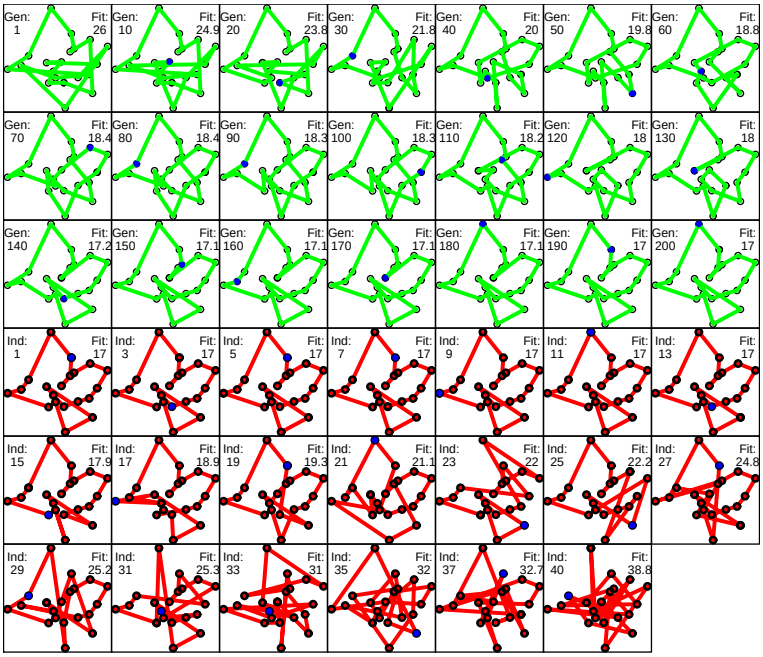
15 ⟨*open graphics device* 15⟩ ≡ ⊂ 7, 17

```
if(!exists("show.graphics") || show.graphics==TRUE){
  h<-ceiling(sqrt(n.interim.plots+n.final.plots))
  x11(); par(mfrow=c(h,h),mar=c(0,0,0,0))
}
```

4 Ein Beispiel

In der folgenden Graphik sehen wir in grün zur Beschreibung des Evolutions-Prozesses die besten Touren verschiedener Generationen. Die Population am Ende der Evolution wird durch die roten Touren beschrieben. Dieses Ergebnis bekommen wir, wenn wir folgenden Chunk aktivieren:

16 ⟨*führe Beispiel aus* 16⟩ ≡
⟨*start genetic algorithm* 7⟩



Als nummerischen Output bekommen wir folgende Werte:

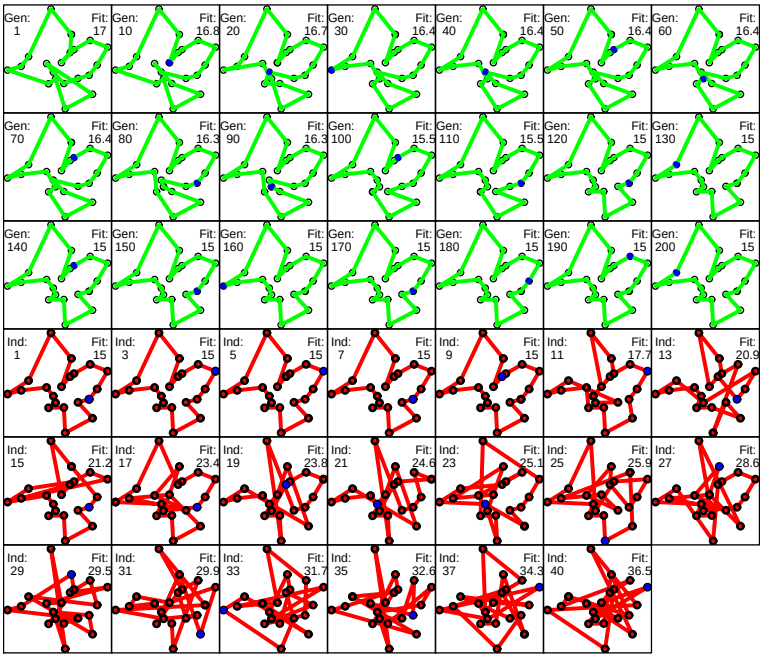
```
beste Tour: 12 2 11 13 14 9 19 8 7 15 16 4 20 10 17 3 5 1 18 6
Strecke: 17.03417
 [1] 17.03417 17.03417 17.03417 17.03417 17.03417 17.03417 17.03417 17.03417
 [9] 17.03417 17.03417 17.03417 17.03417 17.03417 17.05895 17.91501 18.88755
[17] 18.88755 19.25682 19.33007 20.36017 21.12933 21.82894 21.96771 21.98334
[25] 22.19928 23.18567 24.78186 24.98593 25.18656 25.22713 25.25594 25.44864
[33] 31.03602 31.70182 31.96199 32.13252 32.66236 33.51946 35.27894 38.84813
beste Tour: 10 17 3 5 1 6 18 12 2 11 14 13 15 16 19 9 4 8 7 20
```

Weiterführung der Evolution. In dem Beispiel erkennen wir, dass die beste Tour (Individuum 1) noch verbessert werden kann. Deshalb ergänzen wir noch einen Chunk, um die Evolution weiterzutreiben.

17 ⟨*compute some more generations* 17⟩ ≡

```
⟨open graphics device 15⟩
for(counter in 1:n.generations){
  i.generation <- i.generation + 1
  ⟨find fit and sort members 9⟩
  ⟨show interim results 14⟩
  ⟨find new population 10⟩
  ⟨execute some mutations 11⟩
  ⟨execute some crossing-overs 12⟩
}
⟨find fit and sort members 9⟩
⟨show final results 13⟩
```

Weitere `n.generations` Durchgänge führen zur Verbesserung.



In der Tat schlagen sich die optisch erkannten Verbesserungen in den Tourlängen nieder.

```
beste Tour: 10 17 3 5 1 6 18 12 2 11 14 13 15 16 19 9 4 8 7 20
Strecke: 15.00457
 [1] 15.00457 15.00457 15.00457 15.00457 15.00457 15.00457 15.00457 15.00457
 [9] 15.00457 15.71593 17.70607 20.84928 20.88445 21.23569 21.23569 21.23903
[17] 23.41174 23.58455 23.84587 23.84587 24.58094 24.70646 25.14715 25.26953
[25] 25.91298 25.91298 28.59033 29.22695 29.54038 29.81695 29.92904 31.15477
[33] 31.67894 31.67894 32.62843 33.23627 34.31731 35.38771 35.70264 36.51729
```

5 Eine Experimentierfunktion

Für schnelle Experimente soll noch eine Funktion für die Lösung eines TSP verfasst werden. Im Prinzip sieht der Rumpf so aus wie der Grob-Algorithmus, jedoch werden die Parameter zu Funktionsargumenten. Wir fassen sie in einer Tabelle zusammen.

Parameter	Bedeutung
`n.orte`	Anzahl von Zufallsorten
`zz.orte`	Zufallszahlenstart für Zufallsorte
`zz.initial`	Zufallszahlenstart für Startpopulation
`show.graphics`	falls `TRUE` werden Graphiken erstellt
`n.pop`	Populationsgröße
`n.generations`	Anzahl der Generationen, die man verfolgen möchte
`n.stay.alive.anyway`	Anzahl der besten, die auf jeden Fall in der nächsten Epoche weiterleben
`n.drop.off`	Anzahl der schlechtesten, die in einer Epoche entfernt werden
`prob.mutation`	Wahrscheinlichkeit für Mutation eines Individuums
`prob.crossing`	Wahrscheinlichkeit für Crossing-Over für ein Individuum
`n.final.plots`	technischer Parameter, der die Anzahl der dargestellten Individuen am Ende festlegt
`n.interim.plots`	technischer Parameter, der die Anzahl von Plots festlegt, die während der Evolution erstellt werden

18 ⟨*define* `exp.tsp.genetic` 18⟩ ≡ ⊂ 30

```
exp.tsp.genetic <- function(
  n.orte=20, zz.orte=13, zz.initial=17, show.graphics=TRUE,
  n.pop=40, n.generations=200, n.stay.alive.anyway=5, n.drop.off=5,
  prob.mutation=0.2, prob.crossing=0.2, n.final.plots=20, n.interim.plots=20
){
  ⟨define problem 19⟩
  ⟨compute distance matrix 2⟩
  ⟨define some functions 4⟩
  ⟨compute distance matrix 2⟩
  n.alive.by.sampling <- n.pop-n.stay.alive.anyway-n.drop.off
  ⟨initialize population 3⟩
  for(i.generation in 1:n.generations){
    ⟨find fit and sort members 9⟩
    ⟨show interim results 14⟩
    ⟨find new population 10⟩
    ⟨execute some mutations 11⟩
    ⟨execute some crossing-overs 12⟩
  }
  ⟨find fit and sort members 9⟩
  ⟨show final results 13⟩
}
```

Für die Definition unterschiedlicher Probleme ergänzen wir noch:

19 ⟨*define problem* 19⟩ ≡ ⊂ 18

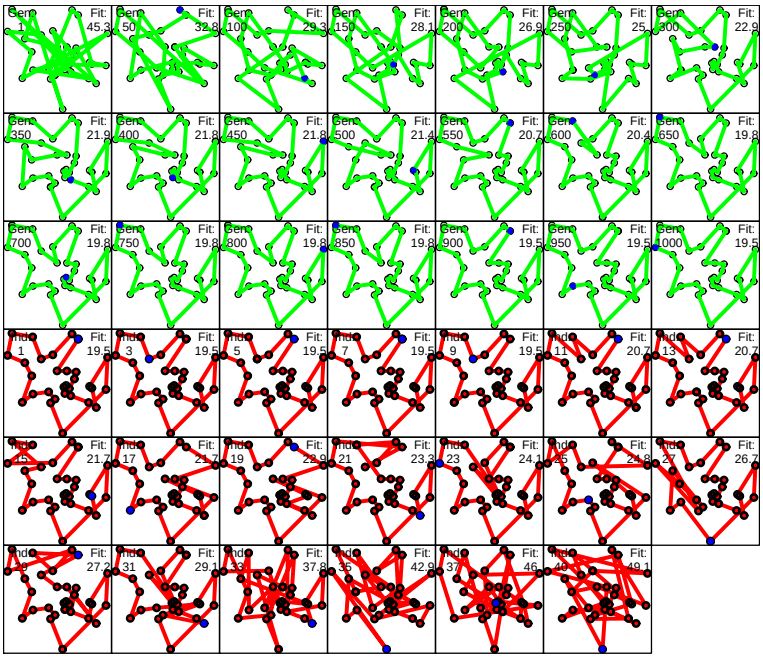
```
set.seed(zz.orte); xy <- cbind(rnorm(n.orte),rnorm(n.orte))
```

Ein kleiner Test rundet die Versuche ab.

20 ⟨*teste of* `exp.tsp.genetic` 20⟩ ≡

```
exp.tsp.genetic(30,n.generations=1000)
```

Na das geht ja ganz einfach!



```
beste Tour: 20 12 2 30 29 14 28 13 11 27 24 15 16 9 18 3 22 5 7 17 10 6 4 26 8 23 1 25 21 19
Strecke: 19.48484
 [1] 19.48484 19.48484 19.48484 19.48484 19.48484 19.48484 19.48484 19.48484
 [9] 19.48484 19.48484 20.65944 20.65944 20.65944 21.21127 21.65918 21.65951
[17] 21.65951 22.36367 22.87375 23.13422 23.34491 23.50500 24.09744 24.10034
[25] 24.76825 25.65508 26.67999 27.21689 27.21689 27.81021 29.09087 34.01907
[33] 37.80533 42.07653 42.86105 45.57328 45.97096 47.89303 48.55550 49.08137
```

6 Weitere Anwendungen

Für das TSP-Problem gibt es eine umfangreiche Liste von Vorgehensvorschlägen. Diese finden in der vorgestellten Vorgehensweise eine interessante Alternative. Doch interessiert uns in diesem Papier primär, welche Idee den genetischen Algorithmen zugrunde liegt und wie die Umsetzung der Idee in einer vorstellbaren Problemsituation sein könnte. Zum Abschluss sollen die Vorstellungen der Leser durch einige weitere Beispiele angeregt werden.

6.1 Variablenselektion

Fragestellung. Im Oktober 2010 hat Javier Trejos in einem Vortrag an der Universität Bielefeld einen genetischen Algorithmus zur Variablen Selektion im Kontext der linearen Regression vorgestellt. Bei dieser Selektionsfrage geht es darum, für eine konkrete Datensituation ein möglichst gutes Regressions-Modell zu finden. Für wachsende Anzahlen von erklärenden Variablen wächst die Anzahl der aufstellbaren Regressions-Modelle gewaltig. Unter Verwendung von 20 Variablen lassen sich schon über 1 Millionen verschiedene Modelle schätzen. Deshalb sind Strategien zum Auffinden geeigneter Modelle bzw. Teilmengen von Variablen höchst interessant. Verbreitet sind Ansätze, bei denen man schrittweise die Anzahl der Variablen vergrößert oder verkleinert, wobei die Entscheidungen anhand eines Kriteriums gefällt werden. Ein Problem solcher Vorgehensweisen besteht jedoch darin, dass eventuell lokale Extrema gefunden werden, die dann zu einem vorschnellen Abbruch des Suchvorgangs führen.

Genetische Algorithmen decken zwar auch in diesem Kontext nicht unbedingt bei begrenztem Ressourceneinsatz das Optimum auf, jedoch findet man regelmäßig recht gute Lösungen. Diese lassen sich dann beispielsweise als Ausgangspunkt weiterer Suchverfahren verwenden.

Vorgehensweise. Notieren wir für eine in einem Modell verwendete Variable eine 1 und für alle anderen eine 0, dann ergibt sich für jedes Modell eine eindeutige 0-1-Folge. Die Population des genetischen Ansatzes wird aus solchen Abfolgen gebildet. Zum Beginn definiert man die Start-Population durch 0-1-Zufallsfolgen. Eine Mutation können wir durch ein Kippen eines Bits in der 0-1-Folge des Genmaterials umsetzen. Die Cross-Over-Operation lässt sich leicht dadurch abbilden, dass von zwei 0-1-Strängen gleich lange Stücke abgeschnitten und ausgetauscht werden. Als wesentliche Parameter gehen in das Verfahren von Trejos ein: Populationsgröße, Wahrscheinlichkeiten für Mutation und Cross-Over und Abbruchbedingungen wie die maximale Iterationszahl. Als Fitness-Kriterium verwendete er das korrigierte Bestimmtheitsmaß R^2_{adjusted}. Zwar besitzen für die Bewertung von Modellen andere Kandidaten Vorzüge (AIC, BIC), doch erfordert die Durchführung des Algorithmus die Auswertung sehr vieler Modelle, so dass die Berechnungszeit für das Kriterium kritisch ist.

6.2 Kurvenanpassung

Den Lösungsansatz der Variablen-Selektion können wir schnell umformen, um eine Modellkurve an eine Punktemenge anzupassen. Dazu ordnen wir jedem Gen einen Modell-Parameter zu und wählen anfangs für ein Mitglied der Startpopulation einen Zufallswert aus. Mutationen lassen sich umsetzen, in dem wir die Werte der Parameter durch Addition positiver oder negativer Werte verändern. Zwei Populationsmitglieder können wir rekombinieren, indem wir Teile ihrer Parameter-Ausstattung austauschen. Als Fitness-Maß kann beispielsweise das bekannte Kleinst-Quadrat-Kriterium oder ein anderes Abstandsmaß zum Einsatz kommen.

Mit diesem Ansatz wollen wir zur Demonstration ein Kreis-Modell finden, das ganz gut durch Punkte verläuft, die wie Gesteinsbrocken auf einem Saturn-Ring

liegen.

Problemfestlegung. Beginnen wir mit der Ziehung einiger Zufallspunkte auf einem gedachten Saturn-Ring. Diese legen wir auf der zweispaltigen Koordinaten-Matrix `xy` ab.

21 ⟨*Kreis: konstruiere Problem* 21⟩ ≡

```
# Anzahl der Datenpunkte
n.pkt <- 100
# Zufallspunkte
set.seed(17); h<-runif(n.pkt)
xy <- cbind(3 + cos(2*pi*h) + runif(n.pkt,-.2,+.2),
            3 + sin(2*pi*h) + runif(n.pkt,-.2,+.2))
```

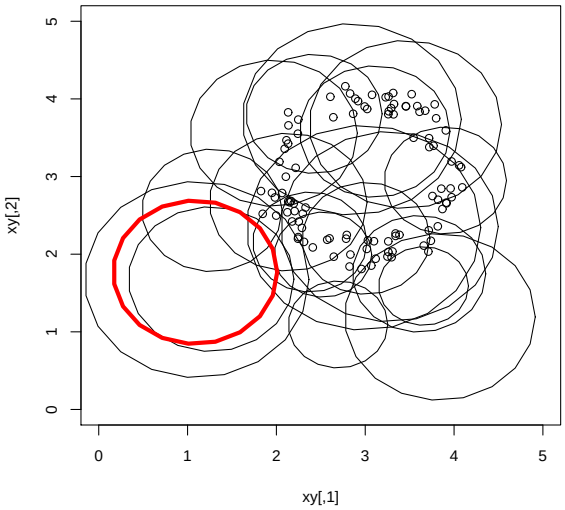
Start-Population. Als Start-Population verwenden wir `m` Kreise, die in der Nähe der Punkte liegen. Die Kreis-Modelle werden beschrieben durch ihre Zentren sowie ihre Radien.

22 ⟨*Kreis: initialisiere Algorithmus* 22⟩ ≡

```
# Anzahl der Modelle
m<-20
# Ziehung von Koordinaten und Radien
set.seed(28)
pop <- cbind(runif(m,1,4),runif(m,1,4),runif(m,.5,1.5))
# Anzahl an Generationen
n.gen <- 50
# Anfangswerte Fitness
fitness.values <- rep(0,m)
# Generations-Counter
i.gen<-1
# Anzahl der besten, die nicht ausgesondert werden
n.best <- 4
# Anzahl der zu mutierenden Individuen
n.mut <- 3
# Verschiebung von Modellen bei Mutation
random.shift <- 0.3
# Kreispunkteanzahl zur Umsetzung der Modelle
n.pkt <- 20
```

⟨*Kreis: zeige Population und Datenpunkte* 26⟩

Folgendes Bild zeigt die Datensituation und die Anfangsmodelle:



Der Grob-Algorithmus. Die Durchführung der Evolution geschieht in einer einfachen Schleife. Zur Vereinfachung verzichten wir auf Rekombinationen. Diese kann der Leser leicht ergänzen, indem er jeweils die Radien zweier Modelle austauscht.

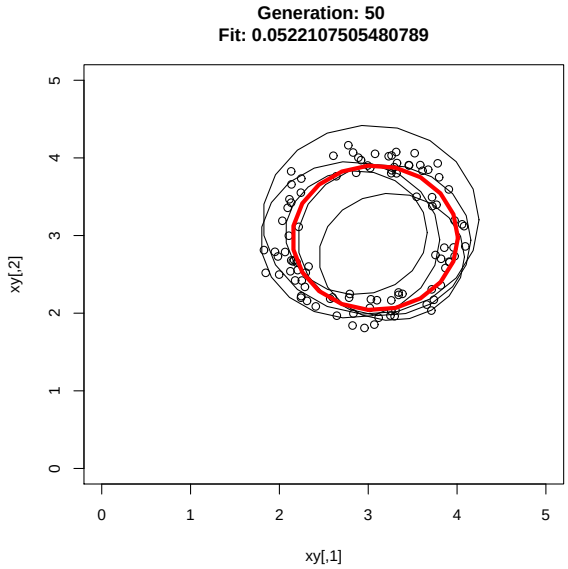
23 ⟨*Kreis: ermittle Generationsfolge* 23⟩ ≡ ⊂ 29

```
for(i.gen in 1:n.gen){
  ⟨Kreis: berechne Fitness-Werte 28⟩
  ⟨Kreis: setze Selektion um 24⟩
  ⟨Kreis: setze Mutation um 25⟩
  ⟨Kreis: zeige Population und Datenpunkte 26⟩
}
⟨Kreis: berechne Fitness-Werte 28⟩
cat("unterschiedliche Modelle am Ende / Fitness:\n")
idx <- c(TRUE,diff(fitness.values)!=0)
print(cbind(x=pop[idx,1],y=pop[idx,2],radius=pop[idx,3],fit=fitness.values[idx]))
```

Nach der 50. Generation enthält die Population noch folgende Modelle:

```
unterschiedliche Modelle am Ende / Fitness:
            x        y    radius        fit
[1,] 3.077705 2.970821 0.9320652 0.05221075
[2,] 3.181139 2.940191 0.9747059 0.04474299
[3,] 2.796680 2.944854 1.0084446 0.04066486
[4,] 2.865279 3.037502 0.7991195 0.03521521
[5,] 3.028124 3.202628 1.2190290 0.03103108
[6,] 3.264067 2.725055 0.8195464 0.02610023
```

Die Population ist etwas degeneriert, da das erste Modell 15 Male identisch enthalten ist. Hier lassen sich schnell Verbesserungen umsetzen. Die folgende Graphik zeigt uns die verbleibenden Modelle. Die rote Kurve repräsentiert das Modell mit den größten Fitness-Wert nach 50 Generationen.



Jetzt gilt es, die Hüllen mit Inhalten zu füllen: Selektion, Mutation und Fitness-Werte müssen umgesetzt werden.

Selektion. Die `n.best` sollen auf jeden Fall überleben, die restlichen Mitglieder werden aus dem Pool mit Wahrscheinlichkeiten proportional zum Fitness-Wert gezogen.

24 ⟨*Kreis: setze Selektion um* 24⟩ ≡ ⊂ 23

```
idx <- sample(1:m,m-n.best,prob=fitness.values/sum(fitness.values),repl=TRUE)
pop.selected <- pop[idx,]; pop.best <- pop[1:n.best,]
pop <- rbind(pop.best,pop.selected)
```

Mutation. Bis auf das beste Element dürfen alle Modelle mutieren. Mutation wird umgesetzt durch Addition einer Zufallsstörung auf die Modellparameter.

25 ⟨*Kreis: setze Mutation um* 25⟩ ≡ ⊂ 23

```
idx<-sample(2:m,n.mut)
pop[idx,] <- pop[idx,]+matrix(runif(n.mut*3,-random.shift,random.shift),ncol=3)
```

Ausgabe. Für die Demonstration ist eine graphische Darstellung vorteilhaft.

26 ⟨*Kreis: zeige Population und Datenpunkte* 26⟩ ≡ ⊂ 22, 23, 29

```
# Darstellung der Punkte
plot(xy,xlim=c(0,5),ylim=c(0,5))
# Darstellung der Modelle
for(i in m:1){
  ⟨Kreis: berechne aus Modell i Kreiskoordinaten 27⟩
  if(i>1) lines(x,y) else lines(x,y,lwd=4,col="red")
}
```

```
if(exists("i.gen")) title(paste("Generation:",i.gen,"\nFit:",fitness.values[1]))
if(0==(i.gen%%5)) cat("Generation:",i.gen,"/ Fits:",fitness.values)
```

Repräsentation der Modelle. Wir wollen ein Kreis-Modell durch eine kleine Anzahl von Punkten auf der Kreislinie repräsentieren. Diese Punkte werden ebenfalls für die Berechnung der Güte des Modells herangezogen.

27 ⟨*Kreis: berechne aus Modell i Kreiskoordinaten* 27⟩ ≡ ⊂ 26, 28

```
hh <- seq(0,1,length=n.pkt)
x <- pop[i,1] + pop[i,3]*cos(2*pi*hh)
y <- pop[i,2] + pop[i,3]*sin(2*pi*hh)
```

Fitness-Werte. Zur Bestimmung der Qualität eines Kreis-Modells suchen wir zu jedem Datenpunkt den nächsten Punkt auf dem Kreis-Modell. Dann addieren wir für jedes Modell die absoluten Koordinatendifferenzen und verwenden den Kehrwert dieser Summe als Fitness-Wert. In einem Zuge sortieren wir die Modelle so, dass das beste Element der Population vorn steht.

28 ⟨*Kreis: berechne Fitness-Werte* 28⟩ ≡ ⊂ 23

```
for(i in 1:m){
  ⟨Kreis: berechne aus Modell i Kreiskoordinaten 27⟩
  fitness.values[i] <- 1/sum( apply(abs(outer(x,xy[,1],FUN="-"))+
                          abs(outer(y,xy[,2],FUN="-")),2,min) )
}
idx <- order(fitness.values,decreasing = TRUE)
pop[] <- pop[idx,]; fitness.values <- fitness.values[idx]
```

Eine Experimentierfunktion. 29 ⟨*definiere* `exp.cirlce.genetic` 29⟩ ≡ ⊂ 30

```
exp.circle.genetic <- function(n.pkt=50, m=20, n.gen=50, seed=28,
                               n.best=4, n.mut=3, random.shift=0.3){
  set.seed(seed); h<-runif(n.pkt)
  xy <- cbind(3 + cos(2*pi*h) + runif(n.pkt,-.2,+.2),
              3 + sin(2*pi*h) + runif(n.pkt,-.2,+.2))
  fitness.values <- rep(0,m); i.gen<-1
  pop <- cbind(runif(m,1,4),runif(m,1,4),runif(m,.5,1.5))
  ⟨Kreis: zeige Population und Datenpunkte 26⟩
  ⟨Kreis: ermittle Generationsfolge 23⟩
}
```

6.3 Verteilungsanpassungen

Einen Vorschlag zur Anpassung eines Verteilungsmodells per genetischem Algorithmus finden wir in Niermann / Jöhnk (2001):

`http://www.wiwi.uni-hannover.de/Forschung/Diskussionspapiere/dp-238.pdf`

Vgl. auch: S. Niermann (2006): Evolutionary Estimation of Parameters of Johnson-Distributions. In: Journal of Statistical Computation and Simulation, 76(3), 185-193.

7 Anhang und weitere Links

Object Index

Code Chunk Index

Weitere Links

`http://www-e.uni-magdeburg.de/harbich/genetische_algorithmen.php`

R-Pakete

Über `http://cran.r-project.org/web/packages/index.html` finden wir folgende Pakete:

Paketname	Kommentar
`gafit`	Genetic Algorithm for Curve Fitting
`galts`	Genetic algorithms and C-steps based LTS estimation
`genalg`	R Based Genetic Algorithm

Start. Zum Abschluss ein kleiner Start-Chunk für den Experimentator.

30 ⟨*start* 30⟩ ≡

```
⟨define exp.tsp.genetic 18⟩ ⟨definiere exp.cirlce.genetic 29⟩
cat("exp.tsp.genetic() definiert!\n")
cat("probiere:\n",sub("NULL","",sub("function","exp.tsp.genetic",
                                    deparse(args(exp.tsp.genetic)))))
cat("probiere:\n",
"exp.circle.genetic(n.pkt=40, m=20, n.gen=50, seed=28,\n
                    n.best=4, n.mut=3, random.shift=0.3)")
```