V9 – 9 Variationen zum Travelling Salesman Problem

File: v9-tsp.rev
in: /home/pwolf/lehre/aud/material

January 15, 2018

1 Einleitung

Immer wieder suchen wir zu einem Problem die beste Lösung. Dieses Vorhaben ist aber oft sehr schwierig oder so zeitaufwändig, dass wir uns dann doch mit einer ganz guten Lösung begnügen. Manchmal ist man sogar froh, überhaupt eine einigermaßen brauchbare Lösung zu finden. Damit man nicht verbohrt an einem Ansatz hängenbleibt, werden in diesem Papier zu einem Problem verschiedene algorithmische Ansatzpunkte vorgeführt.

Ziel dieses Papiers ist es zu demonstrieren, dass es neben einem zunächst betrachteten Weg auch noch ganz andere geben kann. Als Medium wurde das Travelling Salesman Problem gewählt, dass breit bekannt und unmittelbar plausibel ist. Gleichzeitig gehört es zu den schwereren Problemen. Denn es kann bisher nur nicht in polynomialer Zeit gelöst werden. Hier geht es nicht darum, Handelsvertreter mit einer neuen, verbesserten Software auszustatten, sondern den Leser dafür zu sensibilisieren, dass die Menge der möglichen Vorgehensweisen in der Regel vielfältig ist. Wie wir sehen werden, sind aufwändige Algorithmen nicht unbedingt besser. Das ist nicht so schlimm, da ja das Denken in Alternativen im Vordergrund stehen soll.

Folgende Ansätze greifen wir auf:

- **To-The-Next-Algo**: Wir stellen uns vor, wir befinden uns in einem Ort. Dann fahren wir einfach zum nächsten noch nicht besuchten Ort weiter. So verfahren wir, bis wir alle Orte besucht haben und kehren zum Schluss zu unserem Ausgangsort zurück.
- **No-Cross-Algo**: Eine Rundreise, die eine Kreuzung besitzt, kann verbessert werden, indem wir die Kreuzung auflösen. Wir ziehen einfach eine Zufallspermutation und entfernen dann alle Kreuzungen. Durch Wiederholung dieses Vorgehen und Auswahl der besten Lösung erhalten auf einem recht einfachen Weg eine Rundreise.
- **To-The-Next-Algo2**: Wir gehen wie bei dem ersten Algorithmus vor. Dann sorgen wir dafür, dass die Rundreise von Kreuzungen befreit wird.

- **Magnet-Algo**: Stellen wir uns vor, dass wir zu einer Rundreise einen weiteren Ort aufnehmen wollen, der eine Anziehungskraft auf die Pfad-Stücke besitzt. Der Ort zieht das ihm am nächsten gelegene Teilstück am stärksten an und wird in diesem eingebaut. Mit dieser Idee können wir – ausgehend von Zufallspermutationen – Orte entfernen und wieder einbauen. Wir können verschiedene Anfangspermutationen und die Anzahl der Entfernungs-und-Wiedereinbau-Operationen wählen und dann das beste Ergebnis abliefern. Natürlich sollten wir auch hier Kreuzungen auflösen.

- **Max-Dist-Algo**: Bei vielen Problemen hilft der Hinweis: Suche die schwierigste Stelle des Problems und löse dieses zunächst. Für die leichteren wird sich schon eine Lösung finden. So können wir die Orte, deren Distanz zu ihren nächsten Nachbarn sehr groß ist, vordringlich ins Auge fassen. Diese Idee lässt sich operationalisieren, indem wir die Orte als Cluster auffassen und dann das abgelegenste Cluster mit seinem nächsten Nachbarn verschmelzen. Dadurch dass wir jedem Cluster einen Weg durch seine Orte zuordnen, kommen wir zum Schluss zu einer Rundreise.

- **Ameisen-Algo**: Ameisen bilden Straßen zu Futterquellen, indem sie zufällig durch die Landschaft laufen. Wenn sie eine Futterquelle gefunden haben, pendeln sie zwischen Ameisenhaufen und der Quelle. Dabei versprühen sie Duftstoffe (Pheromone) und markieren ihren Weg. In der Nähe gelegene Futterquellen werden wegen des kürzeren Weges häufiger besucht und die zugehörigen Wegstrecken intensiver markiert. Eine verstärkte Markierung erlaubt es anderen Ameisen, die Ausbeute der nahen Futterquelle zu unterstützen und ihrerseits den Geruch zu verstärken. In Übertragung dieses Modells lassen wir Ameisen Rundreisen machen und die Pfadstücke in Abhängigkeit von Weglängen markieren. Kürzere Rundreisen oder Teilstücke erfahren eine höhere Wahrscheinlichkeit von weiteren Ameisen benutzt zu werden. Zum Schluss setzt sich eine Rundreise durch.[1]

- **Bienen-Algo**: Auch Bienen sind darauf angewiesen, gute Futterquellen zu finden und zu ernten. Wir unterstellen dabei das Modell, dass Arbeiter-Bienen zufällig in der Landschaft nach Futter suchen. Wenn sie etwas (eine Blume) gefunden haben, suchen sie in der Nähe nach weiteren Quellen. Beobachter-Bienen merken sich die Ergebnisse der Arbeiter-Bienen und suchen in der Nähe der besten Quellen ihrerseits nach Verbesserungen. Wenn Arbeiter-Bienen eine Futterregion abgearbeitet haben, werden sie zu Pfadfinder-Bienen und machen sich auf, eine neue Region zu entdecken.[2]

- **Kohonen-Algo**: Gemäß der relativen Lage der tastempfindlichen Nervenzellen einer Hand entwickelt sich in unserem Gehirn eine strukturell entsprechende Anordnung von Nervenzellen. Damit besitzt das Gehirn eine neuronale Landkarte für die Hand bzw. für den Sensorraum. Diese

[1] Siehe: `https://de.wikipedia.org/wiki/Ameisenalgorithmus`, `https://ac.els-cdn.com/S0303264797017085/1-s2.0-S0303264797017085-main.pdf?_tid=5a64f6d0-f09a-11e7-82ad-00000aab0f27&acdnat=1514993319_0ae55a66dee379ed78a87855b20b16c9`

[2] Siehe hierzu: `https://roempp.thieme.de/roempp4.0/do/data/RD-02-04169`

Landkarte wird durch den Gebrauch der Hand mit der Zeit ausgebildet (Lernphase) und kann dann verwendet werden, zum Beispiel zur Lokalisation der Stelle eines Mückenstichs. Sogenannte Kohonen-Karten übertragen diese Modellvorstellung in die Rechnerwelt und können insbesondere Lösungen für das Problem des Handlungsreisenden liefern. Dabei werden die Orte als Sensoren interpretiert, die die auf einem Ring angeordneten Neuroren reizen und trainieren. Werden nach der Lernphase die Zuständigkeiten der Neuroren im Raum der Orte repräsentiert, ergibt sich ein Vorschlag für eine Rundreise.[3]

- **Genetik-Algo**: Während der Evolution entstehen durch Mutation und Rekombination von Erbmaterial von Generation zu Generation neue Individuen. An die Umwelt besonders schlecht angepasste sterben aus, gut angepasste haben eine größere Überlebenschance. Wenn wir uns eine Population von Rundreisen vorstellen und als Anpassungsmaß die Weglänge unterstellen, können wir einerseits durch Variation einiger Teilstrecken neue Lösungen schaffen. Wir können andererseits aber auch durch Kombination zweier Lösungen eine neue kreieren. Dieses führt uns zu einem Algorithmus, bei dem eine Menge von Rundreisen durch Aussterben, Mutation und Rekombination Schritt für Schritt verändert wird. Als Ergebnis wird natürlich das am besten angepasste Individum abgeliefert.

Recherchen bei Wikipedia[4] oder bei YouTube[5] zeigen uns noch weitere Ansätze, so dass ein interessierter Leser viel Material für weitere Studien findet.

[3] Weitere Ausführungen sind zu finden unter: `aud/material/kohonen/tsp-2018.rev`
[4] Siehe: `https://de.wikipedia.org/wiki/Problem_des_Handlungsreisenden`
[5] `https://www.youtube.com/watch?v=q6fPk0--eHY`

2 To-The-Next-Algo

Bei diesem ganz einfachen Algorithmus besuchen wir nacheinander alle Orte und bewegen uns von einem Ort zu demjenigen, der nächsten liegt, aber auf der bisherigen Reise noch nicht aufgesucht wurde. Je nach Startort können sich leichte Veränderungen einstellen, so dass wir verschiedene Startorte probieren und das beste Ergebnis abliefern.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `n.runs` :: Anzahl der Läufe
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung

1 ⟨*define* `to.the.next.algo() 1`⟩ ≡ ⊂ 2, 36

```
to.the.next.algo <- function(n.orte = 10, n.runs = 50, seed = 13, seed2 = 13){
  ⟨set seed, towns xy, matrix D, seed2 46⟩
  o.set <- seq(n.orte); fit.best <- Inf; path.best <- o.set
  for( run in 1:n.runs ){  # loop of runs
    path <- sample( o.set, 1 )
    for(i in 2:n.orte){
      act <- path[i - 1]
      idx <- which.min(D[act, -path])
      path <- c(path, o.set[-path][idx])
    }
    fit <- find.dist( D = D, path = path )
    if( fit.best > fit ){  # save improved solution
      fit.best  <- fit; path.best <- path
    }
  }
  path <- path.best
  tit.txt <- "To-The-Next-Algo"; d <- find.dist( D = D, path = path )
  ⟨show path 43⟩  ⟨show towns 42⟩
  ⟨show title tit.txt, fit d and par.txt 45⟩
  return(list(d, path))
}
```

2 ⟨*test* `to.the.next.algo() 2`⟩ ≡

```
n.orte <- 55
n.runs <- 20
seed <- 111; seed2 <- 1
⟨define to.the.next.algo() 1⟩
to.the.next.algo(n.orte = n.orte, n.runs = n.runs, seed = seed, seed2 = seed2)
```

3 No-Cross-Algo

Auch der No-Crossing-Algorithmus ist extrem einfach. In diesem werden nacheinander Zufallswege ermittelt, die dann von Kreuzungen bereit werden.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.runs` :: Anzahl der Läufe
- `n.resolve.crossings` :: Anzahl der Durchläufe zur Auflösung von Pfad-Kreuzungen

3 ⟨*define* `no.cross.algo()` 3⟩ ≡ ⊂ 4, 36

```
no.cross.algo <- function(n.orte = 10, n.runs = 50, seed = 13, seed2 = 13,
                          n.resolve.crossings = 3){
  ⟨set seed, towns xy, matrix D, seed2 46⟩
  path.best <- sample( seq(n.orte) )
  fit.best  <- find.dist( D = D, path = path.best )
  for( run in 1:n.runs ){  # loop of runs
    # choose new permutation
    path <- sample(seq(n.orte))
    for( i in 1:n.resolve.crossings ){ ⟨resolve crossings 41⟩ }
    fit <- find.dist( D = D, path = path )
    if( fit.best > fit ){  # save improved solution
      fit.best  <- fit; path.best <- path
    }
  }
  path <- path.best
  for( i in 1:n.resolve.crossings ){ ⟨resolve crossings 41⟩  }
  tit.txt <- "No-Cross-Algo"; d <- find.dist( D = D, path = path )
  ⟨show path 43⟩  ⟨show towns 42⟩
  ⟨show title tit.txt, fit d and par.txt 45⟩
  return(list(d, path))
}
```

4 ⟨*test* `no.cross.algo()` 4⟩ ≡

```
n.orte <- 55
n.runs <- 100
seed <- 111; seed2 <- 1
⟨define no.cross.algo() 3⟩
no.cross.algo(n.orte = n.orte, n.runs = n.runs, seed = seed, seed2 = seed2)
```

4 To-The-Next-Algo ohne Kreuzungen

Der erste Algorithmus liefert uns sehr oft Lösungen mit Kreuzungen. Wir erhalten deshalb eine Verbesserung, wenn wir die Kreuzungen auflösen.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `n.runs` :: Anzahl der Läufe
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.resolve.crossings` :: Anzahl der Durchläufe zur Auflösung von Pfad-Kreuzungen

5 ⟨*define* `to.the.next.algo2() 5`⟩ ≡ ⊂ 6, 36

```
to.the.next.algo2 <- function(n.orte = 10, n.runs = 50, seed = 13, seed2 = 13,
                              n.resolve.crossings = 3){
  ⟨set seed, towns xy, matrix D, seed2 46⟩
  o.set <- seq(n.orte); fit.best <- Inf; path.best <- o.set
  for( run in 1:n.runs ){  # loop of runs
    path <- sample( o.set, 1 )
    for(i in 2:n.orte){
      act <- path[i - 1]
      idx <- which.min(D[act, -path])
      path <- c(path, o.set[-path][idx])
    }
    for( i in 1:n.resolve.crossings ){ ⟨resolve crossings 41⟩ }
    fit <- find.dist( D = D, path = path )
    if( fit.best > fit ){  # save improved solution
      fit.best  <- fit; path.best <- path
    }
  }
  path <- path.best
  for( i in 1:n.resolve.crossings ){ ⟨resolve crossings 41⟩  }
  tit.txt <- "To-The-Next-Algo2"; d <- find.dist( D = D, path = path )
  ⟨show path 43⟩  ⟨show towns 42⟩
  ⟨show title tit.txt, fit d and par.txt 45⟩
  return(list(d, path))
}
```

6 ⟨*test* `to.the.next.algo2() 6`⟩ ≡

```
n.orte <- 55
n.runs <- 30
seed <- 111; seed2 <- 3
⟨define to.the.next.algo2() 5⟩
to.the.next.algo2(n.orte = n.orte, n.runs = n.runs, seed = seed, seed2 = seed2)
```

5 Magnetismus-Algorithmus

Der Magnetismus-Algorithmus geht von der Vorstellung aus, dass Orte zufällig aus der Rundreise entfernt werden und dann ermittelt wird, an welcher Stelle des verbleibenden Pfades sie am günstigsten eingebaut werden können.

Gehen wir von einer Zufalls-Permutation aus, so stellt sich anfangs fast mit jedem Entfernen und Wieder-Einbauen eine Verbesserung ein. Mit der Zeit wird jedoch die Wahrscheinlichkeit für Verbesserungen immer geringer, so dass wir nach einer bestimmten Zahl von Iterationen den Prozess abbrechen sollten. Jedoch ist das Ergebnis extrem von der Anfangs-Permutation abhängig, so dass wir besser mehrere Anfänge ausprobieren. Den insgesamt besten Pfad liefern wir als Ergebnis ab.

Zum Schluss werden noch Kreuzungen bearbeitet.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `n.runs` :: Anzahl der Läufe
- `iterations.max` :: Anzahl der Durchgänge, bei denen wir jeweils `n.orte` austauschen
- `seed2` :: Zufallsstart für Abwicklung
- `n.resolve.crossings` :: Anzahl der Durchläufe zur Auflösung von Pfad-Kreuzungen am Schluss

7 ⟨*define* `magnet.algo()` 7⟩ ≡ ⊂ 8, 36

```
magnet.algo <- function(n.orte = 10, seed = 13, n.runs = 50, iterations.max = 30,
                        show = FALSE, seed2 = 13, n.resolve.crossings = 1){
  ⟨set seed, towns xy, matrix D, seed2 46⟩
  ⟨set start path 9⟩
  for( run in 1:n.runs ){  # loop of runs
    ⟨choose new permutation 11⟩
    for( i in 1:iterations.max ){
      ⟨choose towns and search neighbors 10⟩
    }
    if( fit.opt > fit.better ){ # save improved solution
      fit.opt  <- fit.better; path.opt <- path.better
    }
  }
  path <- path.opt
  for( i in 1:n.resolve.crossings ){ ⟨resolve crossings 41⟩  }
  tit.txt <- "Magnet-Algo"; d <- find.dist( D = D, path = path )
  ⟨show path 43⟩  ⟨show towns 42⟩
  ⟨show title tit.txt, fit d and par.txt 45⟩
  return(list(d, path))
}
```

8 ⟨*test* `magnet.algo()` 8⟩ ≡

```
# define situation
n.orte <- 10; seed <- 13
n.orte <- 35; seed <- 13
n.runs <- 20; iterations.max <- 10
⟨define magnet.algo() 7⟩
magnet.algo(n.orte = n.orte, seed = seed, n.runs = n.runs,
            iterations.max = iterations.max, seed2 = 77)
```

9 ⟨*set start path* 9⟩ ≡ ⊂ 7

```
path.opt <- path.better <- sample( seq(n.orte) )
fit.opt  <- fit.better  <- find.dist( D = D, path = path.better )
```

Der interessanteste Teil des Algorithmus ist sicher die Entfernung und Replatzierung der Orte. Wiederholt (genauer: `iterations.max` Male) machen wir dabei Folgendes: `n.orte` Male wählen wir einen Ort aus, suchen den nächsten Nachbarn und checken, ob wir den ausgewählten Ort vor oder nach dem Nachbarn in die Rundreise einbauen sollten. Nach jeweils `n.orte` Austauschen stoßen wir die Routine zur Kreuzungsbehebung an.

10 ⟨*choose towns and search neighbors* 10⟩ ≡ ⊂ 7

```
path <- path.better; n.orte.dekr <- n.orte - 1
for( j in 1:n.orte){
  k <- sample(seq(n.orte), 1)      # choose town k
  k.new <- which.min( D[k,] )[1]   # find neighbor k.new
  path <- path [ path != k ]       # remove k
  # put k right of k.new:
  idx <- which( k.new == path )[1]
  path2<- c(path[1:idx], k, if( idx < n.orte.dekr ) path[(idx + 1) : n.orte.dekr ])
  h2 <- find.dist( D = D, path = path2 )
  # put k left of k.new:
  path <- c(if( idx > 1 ) path[(1:(idx-1))], k, path[ (idx ) : n.orte.dekr ])
  fit <- find.dist( D = D, path = path )
  # save the better of the two positions
  if( h2 < fit ) { path <- path2; fit <- h2 }
  if( j == n.orte ){
    ⟨resolve crossings 41⟩
    fit <- find.dist( D = D, path = path )
  }
  # save solution if better
  if( fit < fit.better ){ fit.better <- fit; path.better <- path }
}
```

11 ⟨*choose new permutation* 11⟩ ≡ ⊂ 7

```
path.better <- sample(1:n.orte)
fit.better  <- find.dist( D = D, path = path.better )
```

6 Max-Distance-First

Manchmal ist es hilfreich, die Lösung eines Problems an einer mutmaßlich schwierigen Stelle zu beginnen. Bei dem Handlungsreisendenproblem könnten das weit entfernt liegende Punkte sein. Ausgehend von dieser Überlegung ist der folgende Algorithmus entstanden, in dem wir die Orte zu Beginn in Cluster platzieren und in den Clustern Pfade definieren. Anschließend verschmelzen wir schrittweise Cluster sowie Pfade. Wie bei zwei Fäden gibt es vier Möglichkeiten des Zusammenknotens.

Nach der Aggregation müssen die Distanzen zu den verbleibenden Clustern in Abhängigkeit von den neuen Enden überarbeitet werden. Wieder gäbe es für jedes Cluster-Paar 4 Möglichkeiten der Aggregation, wobei wir natürlich die beste auswählen und so die Distanz jedes Paares bestimmen. Mit dieser Information können wir die Distanz-Matrix der Cluster updaten, welche zu Beginn mit der Distanz-Matrix der Orte übereinstimmt. Wegen der Sicherheit setzen wir auf die Diagonalen einen großen Wert.

Am Ende der Schleife über die Cluster-Aggregation bleiben zwei Cluster übrig, deren Pfade wir zu einer Rundreise zusammenfügen. Hierbei wählen wir aus den Kombinationsmöglichkeiten wieder die beste aus.

Zum Abschluss werden noch Kreuzungen entfernt. Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.resolve.crossings` :: Anzahl der Durchläufe zur Auflösung von Pfad-Kreuzungen am Schluss

12 ⟨*test* `max.distance.algo()` 12⟩ ≡

```
# x11()
# define situation
n.orte <- 25; seed <- 13
n.orte <- 35; seed <- 13
n.orte <- 35; seed <- 14
# n.orte <- 155; seed <- 14

# define parameters
n.runs <- 50; iterations.max <- 30
n.runs <- 10; iterations.max <- 30
⟨define max.distance.algo() 13⟩
max.distance.algo(n.orte = n.orte, seed = seed, 1)
```

13 ⟨*define* `max.distance.algo()` 13⟩ ≡ ⊂ 12, 36

```
max.distance.algo <- function(n.orte = 25, seed = 13, n.resolve.crossings = 2,
                              seed2 = 13){
  ⟨body of max.distance.algo() 14⟩
}
```

14 ⟨*body of* `max.distance.algo()` 14⟩ ≡ ⊂ 13

```
⟨set seed, towns xy, matrix D, seed2 46⟩
DD <- D
path.set <- lapply(1:n.orte, c) # each of the points defines a cluster
for(i in 1:(n.orte - 2)){
  # find cluster being maximal isolated and find its path
  idx.act <- which.max(apply(DD, 1, min))[1]; path.act <- path.set[[idx.act]]
  # find nearest of the other clusters with its path
  idx.neighbor <- which.min(DD[idx.act,])[1]; path.neighbor <- path.set[[idx.neighbor]]
  # find best way to aggregate the two clusters and compose the common path
  h <- cbind(c(h <- path.act[1],      h,
               h <- rev(path.act)[1], h),
             c(h <- path.neighbor[1], hh <- rev(path.neighbor)[1],
               h,                     hh[1]))
  h <- which.min(D[h])[1]
  if( h == 1 ){ path.new <- c( rev(path.neighbor),      path.act        )}
  if( h == 2 ){ path.new <- c(     path.neighbor,       path.act        )}
  if( h == 3 ){ path.new <- c(     path.act,            path.neighbor   )}
  if( h == 4 ){ path.new <- c(     path.act,        rev(path.neighbor)  )}
  # compute distances of the aggregated cluster and the others
  idx.reduced <- -c(idx.neighbor, idx.act)
  h <- path.set[idx.reduced]             # set of other paths
  distances <- pmin(  D[    path.new [1], sapply(h, function(x)     x [1])],
                      D[rev(path.new)[1], sapply(h, function(x)     x [1])],
                      D[    path.new [1], sapply(h, function(x) rev(x)[1])],
                      D[rev(path.new)[1], sapply(h, function(x) rev(x)[1])])
  path.set[[idx.neighbor]] <- path.new  # update set of the nearest neighbor
  path.set <- path.set[ -idx.act ]      # remove isolated cluster being aggregated
  # update distance matrix of clusters
  DD[idx.neighbor, idx.reduced] <- DD[idx.reduced, idx.neighbor] <- distances
  DD <- DD[ -idx.act, -idx.act]; diag(DD) <- 1e5
}
# combine the resulting two clusters
idx.act <-      1; path.act <-      path.set[[idx.act]]      # set path of actual cluster
idx.neighbor <- 2; path.neighbor <- path.set[[idx.neighbor]] # set path of neighbor cluster
h <- cbind(c(h <- path.act[1],      h,
             h <- rev(path.act)[1], h),
           c(h <- path.neighbor[1], hh <- rev(path.neighbor)[1],
             h,                     hh[1]))
# find best way of combining the two clusters
h <- which.min(D[h])[1]
if( h == 1 ){ path.new <- c( rev(path.neighbor),      path.act        )}
if( h == 2 ){ path.new <- c(     path.neighbor,       path.act        )}
if( h == 3 ){ path.new <- c(     path.act,            path.neighbor   )}
if( h == 4 ){ path.new <- c(     path.act,        rev(path.neighbor)  )}
path <- path.new                        # store resulting path in variable "path"
for( i in seq(n.resolve.crossings)){  ⟨resolve crossings 41⟩  }
⟨show path 43⟩  ⟨show towns 42⟩
d <- find.dist( D = D, path = path ); tit.txt = "Max-Dist-Algo"
⟨show title tit.txt, fit d and par.txt 45⟩
return(list(d, path))
```

7 Honig-Bienen-Algorithmus

Beim Bienen-Algorithmen geht man davon aus, dass Bienen an ergiebigen Futterquellen interessiert sind. Dazu organisieren sie ihre Suche so, dass Arbeiter-Bienen bestimmte Gebiete nach den besten Quellen absuchen. Weiterhin gibt es Beobachter-Bienen, die schauen, was die Arbeiter-Bienen gefunden haben, um daraufhin die besten Quellen zu verwalten. Hat eine Arbeiter-Biene ihre Region abgearbeitet, wählt sie zufällig eine neue Region aus, um ihre Suche fortzusetzen.

Wir interpretieren gute Quellen mit kurzen Rundreisen. Andere Rundreisen in dem Gebiet finden wir durch leichte Modifikationen der vorgegebenen Rundreise. Dazu permutieren wir einen Teil der Reise. Die Beobachter-Bienen verwalten die besten Rundreisen der Arbeiter-Bienen und können nahe dieser eigene Suchen unternehmen. Will eine Arbeiter-Biene ein Gebiet wechseln, kann entweder eine völlig neue Permutation der Orte oder es kann als Ausgangspunkt eine bereits als ganz gut eingestufte Reise verwendet werden.

Der Algorithmus verfährt wiederholt (`max.steps` Male) nach folgendem Muster: Im Arbeiter-Bienen-Schritt wird eine bessere Lösung in der Nähe gesucht. Dann wird im Beobachter-Bienen-Schritt eine Arbeiter-Bienen-Lösung ausgewählt und mit einer nahen Alternative verglichen. Die beste Lösung wir jeweils gespeichert. Wenn die Arbeiter-Biene ihre Nahfeld-Erkundungen abgeschlossen hat, wird sie zur Pfadfinder-Biene und sucht sich ein neues Gebiet. Spannend ist natürlich wie die Einzelschritte umgesetzt sind, wobei dem Leser sofort Varianten einfallen werden.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.a.bees` :: Anzahl der Arbeiter-Bienen
- `n.b.bees` :: Anzahl der Beobachter-Bienen
- `max.steps` :: Anzahl der Iterationen über: Arbeiter-, Beobachter- und Pfadfinder-Bienen-Schritt
- `max.tests.a` :: maximale Anzahl vergeblicher Versuche einer Arbeitsbiene ihren Pfad zu verbessern
- `max.steps.a` :: maximale Anzahl der Suchen einer Arbeiter-Biene in einem Arbeiter-Bienen-Schritt
- `len.permute` :: Länge der Teilstücke die permutiert werden

Aus der obigen Beschreibung ergibt sich die folgende grobe Struktur:

15 ⟨*define* `bienen.algo() 15`⟩ ≡ ⊂ 16, 36

```
bienen.algo <- function(n.orte = 25, n.a.bees = 100, n.b.bees = 50, seed = 13, seed2 = 13,
            max.steps = 10, max.tests.a = 5, max.steps.a = 10, len.permute = 25, verbose = FALSE){
  ⟨initialize some objects 26⟩
  ⟨find initial paths for both types of bees 23⟩
  for(step in 1:max.steps){ if(verbose) cat(step, F.b.best) # ... to show that it is going on
    ⟨working bees step: search a better solution in the neighborhood 17⟩
    ⟨observer bees step: choose working bees path, check modification 19⟩
    ⟨store best solution 22⟩
    if( step < max.steps ){
      ⟨explorer bees step: choose new regions for worker bees 21⟩
    }
  }
  ⟨show result of bee algorithm 25⟩
}
```

Der folgende Chunk dient zum Testen der Funktion.

16 ⟨*teste* `bienen.algo() 16`⟩ ≡

```
n.orte <- 35; seed <- 111; seed2 <- 1 # or set seeds: 13 3943
n.a.bees <- 120 # number of worker bees
n.b.bees <-  50 # number of observer bees
max.steps <- 10 # number of iterations
max.tests.a <- 40; max.steps.a <- 60
⟨define bienen.algo() 15⟩
# bienen.algo(n.orte = n.orte, seed = seed, seed2 = 13, n.a.bees = 300, n.b.bees = 150, max.steps = 35)
# bienen.algo(n.orte = n.orte, seed = seed, seed2 = 13, n.a.bees = 120, n.b.bees = 50, max.steps = 10)
# bienen.algo(n.orte = n.orte, n.a.bees = n.a.bees, n.b.bees = n.b.bees, max.tests.a = 10,
#             seed = seed, seed2 = seed2, max.steps = max.steps)
n.orte <- 60; seed <- 111; seed2 <- 1; nab <- 250; nbb <- 50; ms <- 10
bienen.algo(n.orte = n.orte, seed = seed, seed2 = seed2, n.a.bees = nab, n.b.bees = nbb, max.steps = ms,
            max.tests.a = 5, max.steps.a = 10)
```

Zunächst wenden wir uns dem Arbeiter-Bienen-Schritt zu, bei dem verbesserte Lösungen in der Nähe gesucht werden. Die äußere Schleife sorgt dafür, dass alle Bienen zum Zuge kommen. In der inneren versucht jede Biene in ihrer Region die Lösung zu verbessern. `max.tests.a` definiert die Anzahl der Versuche ohne Verbesserung. Ist die Maximalzahl erreicht, gibt die Arbeitsbiene das Gebiet auf.

17 ⟨*working bees step: search a better solution in the neighborhood* 17⟩ ≡ ⊂ 15

```
for( i in seq(n.a.bees)){
  # sol.of.a[i,] stores path of working bee i with fit F.a[i]
  old.path <- sol.of.a[i,]
  for(st.a in 1:max.steps.a){
    ⟨compare old.path with new alternative path and fit f 18⟩
    if( f < F.a[i] ){              # improved paths will be stored
      F.a[i] <- f; sol.of.a[i,] <- old.path <- path; count.a[i] <- 0
    }
    if( max.tests.a < (count.a[i] <- count.a[i] + 1) ) break
  }
} # end of i-loop
```

Um eine bessere Lösung zu finden, permutiert die Biene ausgehend von der Lösung `old.path` ein zufälliges Teilstück der Länge `L`, die zufällig aus `2:len.permute` gezogen wird. Kleine Längen kommen dabei mit höherer Wahrscheinlichkeit zum Zuge. Als Ergebnis erhalten wir den Pfad `path` mit der Fitness `f`.

18 ⟨*compare* `old.path` *with new alternative* `path` *and fit* `f` 18⟩ ≡ ⊂ 17, 19, 20

```
h <- len.permute
L <- sample( 2:h, 1, prob = (h:2) / sum(h:2)); start <- sample(1:n.orte, 1)
idx <- (((start-1) + (0:L)) %% n.orte ) + 1
path <- old.path; path[ idx ] <- sample(path[ idx ])
# improvement by remove crossings: to do
f <- find.dist( D = D, path = path )
```

Im Beobachter-Bienen-Schritt werden die Beobachter-Bienen mit neuen Lösungen ausgestattet. Dazu werden die Lösungen eines Anteil der Arbeiter-Bienen (inklusive der beiden besten) und von den Beobachter-Bienen (inklusive der beiden besten) ausgewählt, wobei der Anteil der Arbeiter-Bienen im Laufe des Berechnungs-Prozesses sinkt. Bessere Lösungen besitzen höhere Ziehungswahrscheinlichkeiten. Zu jeder ausgewählten Lösung wird zusätzlich ein ähnlicher Pfad gesucht und nach Kreuzungsbereinigung verglichen.

19 ⟨*observer bees step: choose working bees path, check modification* 19⟩ ≡ ⊂ 15

```
lambda <- step / max.steps
F.b.h <- F.b; sol.of.b.h <- sol.of.b

# choose n.of.bee.a worker bees including the two best
n.of.bee.a <- floor(n.b.bees * ( 0.75 * lambda + 0.95 * ( 1 - lambda )) )
idx <- which(rank(F.a, ties.method = "random") <= 2) # two best solutions of worker bees
idx.v.a <- c(sample(seq(n.a.bees), n.of.bee.a - 2, prob = (1 / F.a)^2), idx)

for( i in seq(n.of.bee.a) ){
  idx <- idx.v.a[i];  sol.of.b.h[i,] <- old.path <- sol.of.a[idx,]; F.b.h[i] <- F.a[idx]
  ⟨compare old.path with new alternative path and fit f 18⟩
  ⟨resolve crossings 41⟩
  if( f < F.b.h[i] ){
    F.b.h[i] <- f; sol.of.b.h[i,] <- path
  }
}
```

Fortsetzung des letzten Chunks:

20 ⟨*observer bees step: choose working bees path, check modification* 19⟩+ ≡ ⊂ 15

```
# choose n.of.bee.b observer bees including the two best
n.of.bee.b <- n.b.bees - n.of.bee.a
idx <- which(rank(F.b, ties.method = "random") <= 2) # two best solutions of observer bees
idx.v.b <- c(sample(seq(n.b.bees), n.of.bee.b - 2, prob = (1 / F.b)^3), idx)
for( i in seq(n.of.bee.b) ){
  i.b <- i + n.of.bee.a
  idx <- idx.v.b[i];  sol.of.b.h[i.b,] <- old.path <- sol.of.b[idx,]; F.b.h[i.b] <- F.b[idx]
  ⟨compare old.path with new alternative path and fit f 18⟩
  ⟨resolve crossings 41⟩
  if( f < F.b.h[i.b] ){
    F.b.h[i.b] <- f; sol.of.b.h[i.b,] <- path
  }
}

F.b <- F.b.h; sol.of.b <- sol.of.b.h
```

Für den Fall, dass die Arbeiter-Bienen ihr Gebiet abgegrast haben, werden sie zu Pfadfinder-Bienen und machen sich auf die Suche nach einem neuen Gebiet. Andernfalls bleiben sie Arbeiter-Bienen und in ihrer bisherigen Region. Die neue Region bzw. ein neuer Pfad wird entweder durch eine Zufallspermutation oder durch eine Zufallsauswahl aus der Lösungsmenge der Beobachter-Bienen bestimmt.

21 ⟨*explorer bees step: choose new regions for worker bees* 21⟩ ≡ ⊂ 15

```
for( i in seq(n.a.bees) ){
  if( max.tests.a < count.a[i] ){
    count.a[i] <- 0
    if( runif(1) < 0.5 ){
      ⟨choose random permutation path 24⟩
      ⟨resolve crossings 41⟩
      f <- find.dist( D = D, path = path )
    } else {
      idx <- sample( 1:n.b.bees, 1, prob = 1 / F.b^3  )
      path <- sol.of.b[idx,]; f <- F.b[idx]
    }
    sol.of.a[i,] <- path; F.a[i] <- f
  }
}
```

Nach jeder Iteration wird der beste Pfad festgehalten.

22 ⟨*store best solution* 22⟩ ≡ ⊂ 15

```
idx <- which.min(F.b)[1]
if( F.b[idx] < F.b.best ){
  F.b.best <- F.b[idx]; sol.b.best <- sol.of.b[idx,]
}
path <- sol.b.best
⟨resolve crossings 41⟩
sol.b.best <- path; F.b.best <- find.dist( D = D, path = sol.b.best )
```

Zu Beginn müssen alle Bienen mit Lösungen ausgestattet werden. Diese werden auf `sol.of.a` bzw. `sol.of.b` abgelegt. Weiterhin wird für die Beobachter-Bienen der beste Pfade auf `F.b.best` vermerkt. Um nicht zu seltsame Pfad in Erwägung zu ziehen, befreien wir die Anfangslösungen von Kreuzungen.

23 ⟨*find initial paths for both types of bees* 23⟩ ≡ ⊂ 15

```
F.a <- rep(1e10, n.a.bees)
for( i in seq(n.a.bees)){
  ⟨choose random permutation path 24⟩
  ⟨resolve crossings 41⟩
  sol.of.a[i,] <- path; F.a[i] <- find.dist( D = D, path = path )
}

F.b <- rep(1e10, n.b.bees)
for( i in seq(n.b.bees)){
  ⟨choose random permutation path 24⟩
  ⟨resolve crossings 41⟩
  sol.of.b[i,] <- path; F.b[i] <- find.dist( D = D, path = path )
}
idx <- which.min(F.b); F.b.best <- F.b[idx]; sol.b.best <- sol.of.b[idx,]
```

Eine neue Zufallslösung erhalten wir mit `sample()`.

24 ⟨*choose random permutation* `path` 24⟩ ≡ ⊂ 21, 23

```
path <- sample(seq(n.orte))
```

Zum Schluss gilt es noch, dass Ergebnis auszugeben.

25 ⟨*show result of bee algorithm* 25⟩ ≡ ⊂ 15

```
par.txt <- paste(" seeds", seed, ",", seed2,
        "/ n.a.bees", n.a.bees, "/ n.b.bees", n.b.bees, "\n",
        "n =", n.orte, "/ max.steps", max.steps,
        "/ max.tests.a", max.tests.a, "/ max.steps.a", max.steps.a )
path <- sol.b.best; d <- F.b.best; tit.txt <- "Bienen-Algo"
⟨show path 43⟩ ⟨show towns 42⟩
⟨show title tit.txt, fit d and par.txt 45⟩
return(list(d, path))
```

Die Matrizen `sol.of.a` und `sol.of.b` nehmen für die Bienen der beiden Bienen-Typen zeilenweise eine Lösung auf.

26 ⟨*initialize some objects* 26⟩ ≡ ⊂ 15

```
len.permute <- min(len.permute, ceiling(n.orte/2))
sol.of.a <- matrix( NA, n.a.bees, n.orte)
sol.of.b <- matrix( NA, n.b.bees, n.orte)
count.a <- rep(0, n.a.bees)
cols <- rainbow(max.steps)
⟨set seed, towns xy, matrix D, seed2 46⟩
```

8 Ameisen-Algorithmus

Die Grundidee des Ameisen-Algorithmus besteht darin, dass die Ameisen sich zufällig einen Pfad durch die Orte suchen. In jedem Ort wird der nächste Ort jedoch gemäß einer Wahrscheinlichkeitsfunktion gewählt. Die Wahrscheinlichkeiten hängen von zwei Aspekten ab: Erstens von der Weglänge des Segments und zweitens wie häufig des Segment an kurzen Pfaden beteiligt war. Aus Ameisensicht lässt sich vorstellen, dass auf jedem Teilstück eine Ameise einen Duftstoff versprüht, wobei die Stärke mit der Pfadlänge aber auch mit der Segmentlänge sinkt. Hohe Duftdosen wirken sich in hohen Auswahlwahrscheinlichkeiten aus.

In der vorliegenden Umsetzung wurde noch Dämpfungsterme in Abhängigkeit von `i/n.ameisen` integriert, so dass die ersten Ameisen eine nicht zu große Wirkung haben sollten. Zum Ende des Ameisenstromes sollten sich jedoch die Wahrscheinlichkeitsverteilungen gegen die Ein-Punkte-Verteilung bewegen.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `param` :: zwei Faktoren, um den Einfluss von Segment- und der Pfad-Länge multiplikativ zu steuern
- `n.ameisen` :: maximale Anzahl von Ameisen
- `n.resolve.crossings` :: Anzahl der Durchläufe zur Auflösung von Pfad-Kreuzungen am Schluss

27 ⟨*test* `ameisen.algo() 27`⟩ ≡

```
# x11()
# define situation
n.orte <- 35; seed <- 222; seed2 = 19

# define parameters
# n.runs <- 50; iterations.max <- 30
⟨define ameisen.algo() 28⟩
ameisen.algo(n.orte = n.orte, seed = seed, seed2 = seed2, param = c(.1,.1))
```

28 ⟨*define* `ameisen.algo() 28`⟩ ≡ ⊂ 27, 36

```
ameisen.algo <- function(n.orte = 25, seed = 13, seed2 = 13, param = c(.1,.1),
                         n.resolve.crossings = 1, n.ameisen = 2000, verbose = FALSE){
  ⟨body of ameisen.algo() 29⟩
}
```

29 ⟨*body of* `ameisen.algo()` 29⟩ ≡ ⊂ 28

```
⟨set seed, towns xy, matrix D, seed2 46⟩
DD <- D
# define matrix of probabilities where to go next
pm <- D * 0  + 1 / (n.orte - 1)       # rows contain distributions
# define best solution up to now
path.best <- path <- 1:n.orte; fit.best <- find.dist( D = D, path = path )
# process ants:
for(i in 1:n.ameisen){
  path[1] <- sample(seq(n.orte), 1) # choose starting point of ant i
  for(j in 2:n.orte){               # find points of the path
    p <- pm[path[j-1],]             # distribution for next points after visiting point (j-1)
    p[ path[1:(j-1)] ] <- 0         # deleting probabilities for towns already visited
    path[j] <- sample( seq(n.orte), 1, prob = p) # choose points
  }
  ⟨resolve crossings 41⟩  ⟨resolve crossings 41⟩       # resolve crossings of path
  d <- find.dist( D = D, path = path );             # find length of path
  idx.mat <- cbind(path, c(path[-1], path[1]))      # generate matrix to describe line segments
  fac.1 <- (1 + param[1] / d^2          * (i / n.ameisen)^2) # factor of length of whole path
  fac.2 <- (1 + param[2] / D[ idx.mat ] * (i / n.ameisen)^1) # factor of segment lengths
  pm[ idx.mat ] <- pm[ idx.mat ] * (fac.1 * fac.2)  # modify relevant seqment probabilities
  # scale row distributions to reduce numerical problems
  p.lim <- apply( pm, 1, max) / 10000; pm[] <- pmax(pm, p.lim); pm <- pm / rowSums(pm)
  if( d < fit.best ){ fit.best <- d; path.best <- path }     # save best solution
  if( min(apply( pm, 1, max)) > 0.97 ){if(verbose)print(i);break} # if probabilities converge stop
}
path <- path.best
for( i in 1:n.resolve.crossings ){  ⟨resolve crossings 41⟩  }
d <- find.dist( D = D, path = path )
⟨show path 43⟩   ⟨show towns 42⟩
tit.txt <- "Ameisen-Algo"; par.txt <- paste("param :", paste(param, collapse = ","))
⟨show title tit.txt, fit d and par.txt 45⟩
return(list(d, path))
```

9 Kohonen-Algorithmus

Die Implementierung ist an anderer Stelle – s.o. – beschrieben und hat mit geringen Modifikationen zum Beispiel der Argumentnamen zu folgender Definition geführt.

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.runs` :: Anzahl der Schritte in der Lernphase
- `n.neuronen` :: Anzahl der Neuronen
- `eps` :: Faktor für den Lernprozess
- `n.plots` :: Steuerung der Anzahl der graphischen Zwischenergebnisse
- `wait` :: Steuerung der Wartezeit zwischen der Ausgabe von Zwischenergebnissen

30 ⟨*define* `kohonen.algo()` 30⟩ ≡ ⊂ 32, 36

```
kohonen.algo <- function(n.orte = 10, n.runs = 50, n.neuronen = 100, seed = 13,
                         seed2 = 1, eps = 0.8, n.plots = 10, wait = 0.005)
{
    ⟨body of kohonen.algo() 31⟩
}
```

31 ⟨*body of* `kohonen.algo()` 31⟩ ≡ ⊂ 30

```
⟨set seed, towns xy, matrix D, seed2 46⟩
set.seed(seed2)
r <- 1:n.neuronen
ek <- (2 * pi) * r/n.neuronen
gew <- gew.old <- 0.5 + 0.25 * cbind(sin(ek), cos(ek))
rortnr <- sample(1:n.orte, n.runs, replace = TRUE)
dump.set <- round(seq(1, n.runs, length = (n.plots + 1)))
for (i in 1:n.runs) {     #   print(i)
    idx <- rortnr[i]
    v <- xy[idx, ]
    v.mat <- matrix(v, n.neuronen, 2, byrow = TRUE)
    d <- gew - v.mat
    abstand.q <- apply(d * d, 1, "sum")
    rstrich <- which.min(abstand.q)
    sigma.q <- (50 * 0.02^(i/n.runs))^2
    reiz.zentrum <- pmin(abs(r - rstrich), abs(r - (rstrich - n.neuronen)))
    h <- exp(-0.5 * reiz.zentrum^2/sigma.q)
    hrr.str <- cbind(h, h)
    gew <- gew + eps * (v.mat - gew) * hrr.str
    if (i %in% dump.set) {
        par(col = "white"); lines(gew.old); par(col = 1)
        lines(gew[, 1], gew[, 2]); gew.old <- gew
    }
    Sys.sleep(wait)
}
points(gew[, 1], gew[, 2], pch = 2, col = "blue") # positions of neurons
idx <- rep(0, n.orte)
for (i in 1:n.orte) {
    idx[i] <- which.min(sqrt((gew[, 1] - xy[i, 1])^2 +
                             (gew[, 2] - xy[i, 2])^2))[1]
}
path <- seq(n.orte)[order(idx)]; d <- find.dist( D = D, path = path)
⟨show path 43⟩   ⟨show towns 42⟩
tit.txt <- "Kohonen-Algo"; par.txt <- paste("n Neuronen :", n.neuronen)
⟨show title tit.txt, fit d and par.txt 45⟩
list(d, path)
```

32 ⟨*test* `kohonen.algo()` 32⟩ ≡

```
n.orte <- 20
n.runs <- 20
⟨define kohonen.algo() 30⟩
kohonen.algo(n.orte = n.orte, n.runs = 5*n.runs, seed = seed, seed2 = seed2)
```

10 Genetische Problemlösungen

Auch dieser Algorithmus ist andernorts beschrieben und etwas modifiziert worden. [6]

Als Parameter ergeben sich:

- `n.orte` :: Zahl der Orte
- `seed` :: Zufallsstart für Ziehung der Orte
- `seed2` :: Zufallsstart für Abwicklung
- `n.pop` :: Populationsgröße
- `n.generations` :: Anzahl der Generationen
- `n.stay.alive.anyway` :: Prozentsatz der auf jeden Fall überlebenden
- `n.drop.off` :: Prozentsatz der auf jeden Fall sterbenden
- `prob.mutation` :: Mutationswahrscheinlichkeit
- `prob.crossing` :: Wahrscheinlichkeit für Rekombination

33 ⟨*define* `genetic.algo()` 33⟩ ≡ ⊂ 35, 36

```
genetic.algo <- function (n.orte = 20, seed = 13, seed2 = 17,
    n.pop = 40, n.generations = 200, n.stay.alive.anyway = 5,
    n.drop.off = 5, prob.mutation = 0.2, prob.crossing = 0.2){
  ⟨body of genetic.algo() 34⟩
}
```

[6] Suche: `genetic-algos.pdf`

34 ⟨*body of* genetic.algo() 34⟩ ≡ ⊂ 33

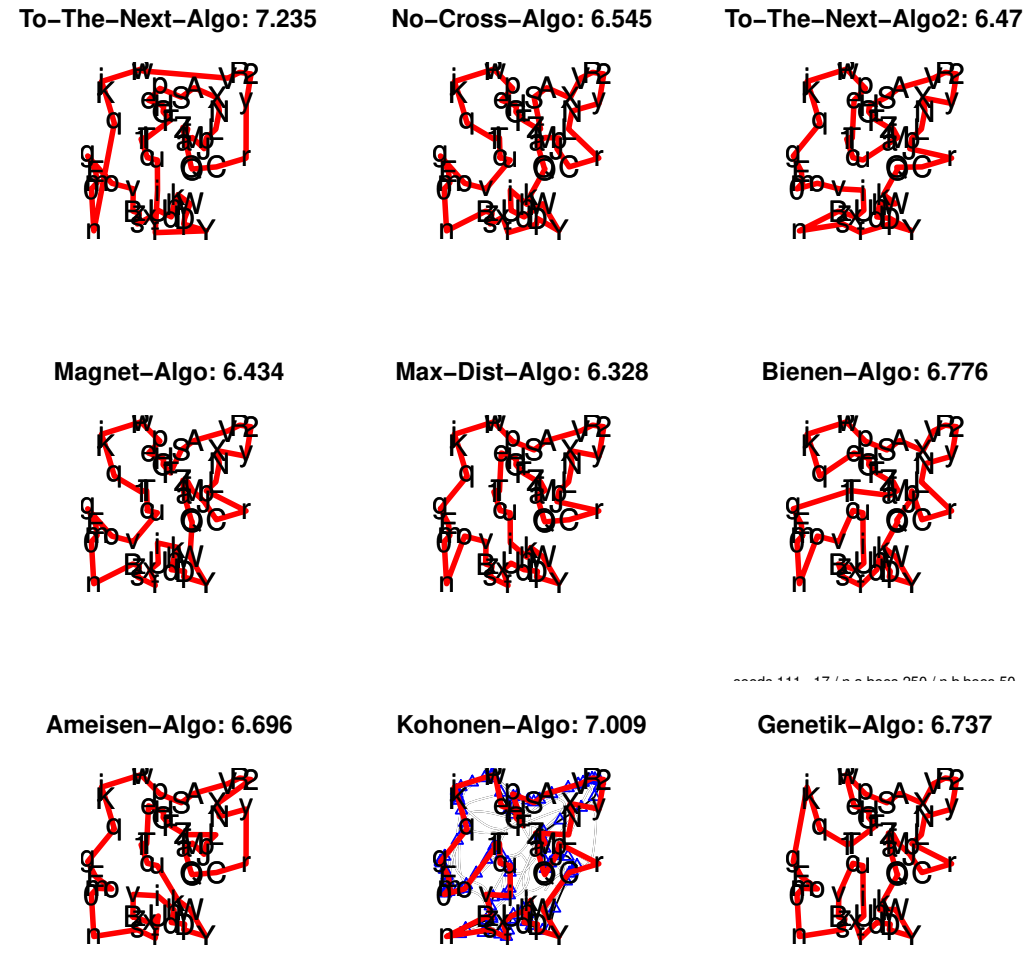
```
⟨set seed, towns xy, matrix D, seed2 46⟩
set.seed(seed2)
create.random.individuals <- function(x) sample(1:n.orte)
prob.stay.alive <- function() fits/sum(fits)
fitness <- function(members) {
    fit <- 1/sum(D[cbind(members, c(members, members[1])[-1])]); fit
}
n.alive.by.sampling <- n.pop - n.stay.alive.anyway - n.drop.off
pop <- sapply(1:n.pop, create.random.individuals)
for (i.generation in 1:n.generations) {
    fits <- apply(pop, 2, fitness); idx <- order(fits, decreasing = TRUE)
    pop <- pop[, idx];              fits <- fits[idx]
    l.ways <- 1/fits;               pop.new <- pop[, 1:n.stay.alive.anyway]
    pop.new <- cbind(pop.new, sapply(1:n.drop.off, create.random.individuals))
    idx.alive <- sample(1:n.pop, n.alive.by.sampling, TRUE, prob = prob.stay.alive())
    pop <- cbind(pop.new, pop[, idx.alive])
    n.mut <- rbinom(1, n.pop, prob.mutation)
    for (i in seq(n.mut)) {
        k <- sample(2:n.pop, 1)
        two.pos <- sample(1:n.orte, 2)
        pop[two.pos, k] <- pop[two.pos[2:1], k]
    }
    n.crossing <- rbinom(1, n.pop, prob.crossing)
    for (i in if (n.crossing > 0) 1:n.crossing else NULL){
        k <- sample(2:n.pop, 2);     j <- sample(1:n.orte, 1)
        p1 <- pop[, k[1]];           p2 <- pop[, k[2]]
        tail1 <- p1[j:n.orte];       tail2 <- p2[j:n.orte]
        p1 <- p1[-match(tail2, p1)]; p2 <- p2[-match(tail1, p2)]
        p1 <- c(tail2, p1);          p2 <- c(tail1, p2)
        pop[, k] <- cbind(p1, p2)
    }
}
fits <- apply(pop, 2, fitness)
idx <- order(fits, decreasing = TRUE)
pop <- pop[, idx]
fits <- fits[idx]
l.ways <- 1/fits #; cat("beste Tour:", pop[, 1]); cat("Strecke:", l.ways[1])
path <- pop[,1]
⟨resolve crossings 41⟩; ⟨resolve crossings 41⟩; ⟨resolve crossings 41⟩
d <- find.dist( D = D, path = path )
⟨show path 43⟩  ⟨show towns 42⟩
tit.txt <- "Genetik-Algo"; par.txt <- paste("n Individuen :", n.pop)
⟨show title tit.txt, fit d and par.txt 45⟩
list(d, path)
```

35 ⟨*test* genetic.algo() 35⟩ ≡

```
n.orte <- 20
n.runs <- 20
seed <- 111; seed2 <- 1
n.pop <- 100
n.generations <- 500
n.stay.alive.anyway <- 10
n.drop.off <- 5
prob.mutation <- 0.2
prob.crossing <- 0.2
⟨define genetic.algo() 33⟩
genetic.algo(n.orte = n.orte, seed = seed, seed2 = seed2, n.pop = n.pop,
    n.generations = n.generations, n.stay.alive.anyway = n.stay.alive.anyway,
    n.drop.off = n.drop.off, prob.mutation = prob.mutation, prob.crossing = prob.crossing)
```

11 Vergleich der Algorithmen

Ein Vergleichsergebnis zeigt folgendes Bild



Zu den Rundreisen durch die 55 Orte gehören folgende Pfadlängen:

	UpToNear	UpToNear2	NoCross	Magnet	MaxDist	Ameisen	Bienen	Kohonen	Genetik
55	7.235	6.47	6.545	6.434	6.328	6.696	6.776	7.009	6.737

Mit folgendem Code wurden die obigen Ergebnisse erzielt.

36 ⟨*compare algorithms* 36⟩ ≡

```
# some settings
seed <- 111; seed2 <- 17; n.runs <- 20; iterations.max <- 10; nc <- 3; param <- c(.1, .1)
nab <- 250; nbb <- 50; ms <- 10; n.ameisen <- 2000; max.tests.a <- 5; max.steps.a <- 10
# seed2 <- as.numeric(substring(gsub("[^0-9]","",date()),5,8))
n.pop <- 300;    n.generations <- 1500; n.stay.alive.anyway <- 10
n.drop.off <- 5; prob.mutation <- 0.2;  prob.crossing <- 0.2

⟨define magnet.algo() 7⟩
⟨define max.distance.algo() 13⟩
⟨define ameisen.algo() 28⟩
⟨define bienen.algo() 15⟩
⟨define to.the.next.algo() 1⟩
⟨define to.the.next.algo2() 5⟩
⟨define no.cross.algo() 3⟩
⟨define kohonen.algo() 30⟩
⟨define genetic.algo() 33⟩

n.orte.set <- 5*(4:20); n.orte.set <- 55
result <- matrix(0, length(n.orte.set), 9)
colnames(result) <- c("ToNext", "NoCross", "ToNext2", "Magnet", "MaxDist",
                      "Bienen", "Ameisen", "Kohonen", "Genetik")
rownames(result) <- as.character(n.orte.set)
old.par <- par(mfrow = c(3,3), mai = c(0.5, 0, 0.5, 0), omi = c(0,0,0,0))
⟨do some tests 37⟩
par(old.par); par(mfrow = c(1,1)); print( round(result, 3) ); NULL
```

37 ⟨*do some tests* 37⟩ ≡ ⊂ 36, 39

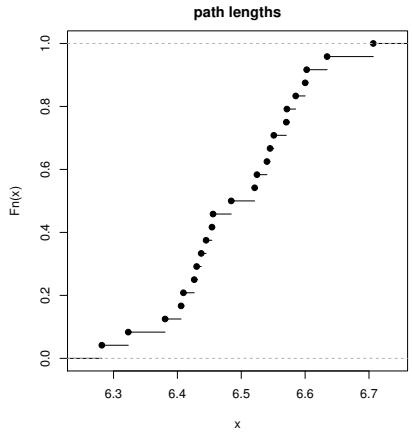
```
for( i in seq(along = n.orte.set) ){ cat(i, "aus {", n.orte.set, "}")
  n.orte <- n.orte.set[i]
  A <- to.the.next.algo(n.orte = n.orte, n.runs = n.runs, seed = seed, seed2 = seed2)[[1]]
  B <- no.cross.algo(n.orte = n.orte, n.runs = 5*n.runs, seed = seed, seed2 = seed2)[[1]]
  C <- to.the.next.algo2(n.orte = n.orte, n.runs = n.runs, seed = seed, seed2 = seed2)[[1]]
  D <- magnet.algo(n.orte = n.orte, seed = seed, n.runs = n.runs, iterations.max = iterations.max,
                   n.resolve.crossings = nc)[[1]]
  E <- max.distance.algo(n.orte = n.orte, seed = seed, n.resolve.crossings = nc, seed2 = seed2)[[1]]
  F <- bienen.algo(n.orte = n.orte, seed = seed, seed2 = seed2, n.a.bees = nab, n.b.bees = nbb,
                   max.tests.a = max.tests.a, max.steps.a = max.steps.a, max.steps = ms)[[1]]
  G <- ameisen.algo(n.orte = n.orte, seed = seed, seed2 = seed2, param = param, n.resolve.crossings = nc,
                    n.ameisen = n.ameisen)[[1]]
  H <- kohonen.algo(n.orte = n.orte, n.runs = 20*n.runs, seed = seed, seed2 = seed2)[[1]]
  I <- genetic.algo(n.orte = n.orte, seed = seed, seed2 = seed2, n.pop = n.pop,
    n.generations = n.generations, n.stay.alive.anyway = n.stay.alive.anyway,
    n.drop.off = n.drop.off, prob.mutation = prob.mutation, prob.crossing = prob.crossing)[[1]]
  result[i,] <- c(A, B, C, D, E, F, G, H, I); # cat(i, result[i,])
}
```

Natürlich hängen die Ergebnisse vom Zufall ab. Das wollen wir mit folgendem Experiment einmal fokussieren:

38 ⟨*ein Experiment zur Demonstration der Zufallsabhänigkeit* 38⟩ ≡

```
n.orte.set <- 55; n.runs <- 20; seed = 111; seed2.set <- 1:24
res <- rep(0, length(seed2.set))
old.par <- par(mfrow = c(h <- ceiling(sqrt(length(seed2.set))),h),
               mai = c(0.5, 0, 0.5, 0), omi = c(0,0,0,0))
for(i in seq(along = seed2.set)){ cat(i, "aus {", seed2.set, "}")
  res[i] <- no.cross.algo(n.orte = n.orte, n.runs = 5*n.runs, seed = seed, seed2 = seed2.set[i])[[1]]
}
par(old.par); par(mfrow = c(1,1), mai = c(1, 1, 0.5, 0))
plot( ecdf(res), main = "path lengths" )
```

Die Verteilung der Rundreiselängen, die der Magnet-Algorithmus in Abhängigkeit von `seed2` erarbeitet, zeigt folgende empirische Verteilungsfunktion.



Als letztes Experiment wollen wir noch checken, wie die Algorithmen bei unterschiedlichen Ortsanzahlen abschneiden.

39 ⟨*Experimente mit unterschiedlichen Ortsanzahlen* 39⟩ ≡
```
n.orte.set <- 5*(4:20); result <- matrix(0, length(n.orte.set), 9)
colnames(result) <- c("ToNext", "NoCross", "ToNext2", "Magnet", "MaxDist",
                      "Bienen", "Ameisen", "Kohonen", "Genetik")
rownames(result) <- as.character(n.orte.set)
old.par <- par(mfrow = c(3,3), mai = c(0.5, 0, 0.5, 0), omi = c(0,0,0,0))
⟨do some tests 37⟩
par(old.par); par(mfrow = c(1,1)); NULL
```

40 ⟨*Experimente mit unterschiedlichen Ortsanzahlen* 39⟩+ ≡
```
options(width = 100)
cbind(data.frame(round(result,2)),
      Gewinner=colnames(result)[ apply(result, 1, function(x) which.min(x)[1]) ])
```

Wir erhalten folgende Ergebnismatrix, bei der die besten Einträge mit + hervorgehoben sind:

	ToNext	NoCross	ToNext2	Magnet	MaxDist	Bienen	Ameisen	Kohonen	Genetik	Gewinner
20	4.44	+3.79	+3.79	+3.79	3.88	+3.79	+3.79	3.82	+3.79	NoCross
25	4.29	3.87	3.93	3.88	3.92	4.00	+3.84	4.59	3.99	Ameisen
30	5.36	4.92	4.92	+4.86	5.07	5.01	+4.86	5.29	4.92	Magnet
35	5.61	5.24	5.31	5.26	5.31	5.36	+5.23	5.46	5.56	Ameisen
40	6.57	6.07	5.91	5.97	6.23	6.01	+5.88	6.20	5.95	Ameisen
45	6.19	5.90	+5.66	+5.66	5.84	5.95	5.82	6.03	5.94	ToNext2
50	6.15	6.01	+5.94	5.95	6.29	6.23	5.98	6.33	6.11	ToNext2
55	7.24	6.55	6.47	6.43	+6.33	6.78	6.70	7.01	6.74	MaxDist
60	7.82	6.58	6.58	+6.35	6.84	6.56	6.50	6.75	6.74	Magnet
65	7.41	6.83	6.82	6.73	7.08	7.26	+6.68	7.17	7.82	Ameisen
70	7.29	6.49	+6.36	6.38	7.21	7.10	6.50	6.95	7.07	ToNext2
75	7.73	6.98	+6.85	6.89	7.36	7.62	7.12	7.64	7.63	ToNext2
80	7.85	7.52	7.49	+7.40	7.84	7.68	7.89	8.70	8.29	Magnet
85	8.92	8.31	7.83	+7.73	8.26	8.29	8.10	8.70	8.52	Magnet
90	8.87	7.73	7.60	+7.49	8.07	8.37	8.06	8.13	8.38	Magnet
95	9.60	8.00	7.90	+7.83	8.04	8.24	8.20	9.03	8.54	Magnet
100	9.70	8.36	8.37	+8.16	8.87	8.50	8.53	9.32	8.78	Magnet

12 Gemeinsame Elemente

Bei vielen Algorithmen können Pfade entstehen, die Kreuzungen enthalten. Durch Auflösung dieser stellen sich Verbesserungen ein. Da dieser Chunk von den Variablen `n.orte, D, path` ausgeht, müssen ggf. bei seiner Einbettung kleine Anpassungen vorgenommen werden.

41 ⟨*resolve crossings* 41⟩ ≡ ⊂ 3, 5, 7, 10, 14, 19, 20, 21, 22, 23, 29, 34

```
for( del in 2:floor(n.orte/2) ){
  for( p in 1:n.orte ){
    pp  <- p + del; if( n.orte < pp  ) pp  <- pp  - n.orte
    pn  <- p + 1;   if( n.orte < pn  ) pn  <- pn  - n.orte
    ppn <- pp + 1;  if( n.orte < ppn ) ppn <- ppn - n.orte
    if( ( D[path[p ], path[pn ]] +   # segment of old path
          D[path[pp], path[ppn]]  )  # other segment
          >
        ( D[path[p ], path[pp ]] +   # segment of old path
          D[path[pn], path[ppn]]  )  # other segment
    ){                               # improvement by change:
      idx <- if( pn < pp ) pn:pp else c(pn:n.orte, 1:pn)
      path[ idx ] <- rev( path[ idx ] )
    }
  }
}
```

42 ⟨*show towns* 42⟩ ≡ ⊂ 1, 3, 5, 7, 14, 25, 29, 31, 34

```
points(xy, pch = symb, cex = 1.5)
```

43 ⟨*show* `path` 43⟩ ≡ ⊂ 1, 3, 5, 7, 14, 25, 29, 31, 34

```
lines( xy[ c(path, path[1]), ], lwd = 3, col = "red")
```

44 ⟨*define* `find.dist()` 44⟩ ≡ ⊂ 46

```
find.dist <- function(xy, D, path){
  if( missing(D) ){
    xy <- rbind( xy, xy[1,])
    fit <- sum( sqrt(  diff(xy[,1])^2 + diff(xy[,2])^2 ) )
  } else {
    if( path[1] != path[length(path)] ) path <- c(path, path[1])
    idx.mat <- cbind( path[ -length(path) ], path[-1])
    fit <- sum( D[ idx.mat ] )
  }
  fit
}
```

45 ⟨*show title* `tit.txt`, *fit* `d` *and* `par.txt` 45⟩ ≡ ⊂ 1, 3, 5, 7, 14, 25, 29, 31, 34

```
title( paste(tit.txt, round(d, 3), sep = ": ") )
if(exists("par.txt")) title( sub = par.txt, cex.sub = 0.7 )
```

46 ⟨*set seed, towns* `xy`, *matrix* `D`, *seed2* 46⟩ ≡ ⊂ 1, 3, 5, 7, 14, 26, 29, 31, 34

```
set.seed(seed)
xy <- cbind( x = runif(n.orte), y = runif(n.orte) )
⟨define find.dist() 44⟩
symb <- c(letters, LETTERS, 0:9)
# par(pty = "s")
plot(xy, axes = FALSE, pch = symb, xlab = "", ylab = "", cex = 1.5)
D <- as.matrix(dist(xy)); diag(D) <- 1e100
set.seed(seed2)
```

13 Anhang

Object Index

Code Chunk Index