UiO : Matematisk institutt
Det matematisk-naturvitenskapelige fakultet

STK-4051/9051 Computational Statistics Spring 2021
Examples IRLS & Chapter 3

Instructor: Odd Kolbjørnsen, oddkol@math.uio.no

Last time

- Iterative reweighted least squares (IRLS)
- Method of moments (constrained optimization)
- Alternating Direction Method of Multipliers (ADMM)
- Heuristics: Algorithms that find a good local optima within tolerable time
  - Local search
  - Simulated annealing
  - Tabu algorithm
  - Genetic algorithm

Question

- From what I understand iteratively reweighed least squares is used when the data points have varying quality, that you are able to weight points the are likely more reliable, higher than others.

But I am struggling to see how it actually works. I have read 2.2.1.1 in the book but am struggling to understand it.

1. Would you be able to revisit or any tips, or have another resource that I could look at?
   - Literature on GLM
   - http://www.sfu.ca/~lockhart/richard/350/08_2/lectures/GLMTheory/web.pdf
   - Literature on Lp norm
   - https://en.wikipedia.org/wiki/Iteratively_reweighted_least_squares
   - http://sepwww.stanford.edu/data/media/public/docs/sep115/jun1/paper_html/node2.html
   - Tip: implement an example (or look at one)

2. How much of the details are we expected to know?
   - You should be abel to use it in an example when asked to do so. (you will get hints to progress)

Example: L1-regression robustness to outliers

- Quadratic loss:

$$\arg\min_{\beta} \sum_{i=1}^{n} (y_i - \beta^T x_i)^2$$

- Absolute loss:

$$\arg\min_{\beta} \sum_{i=1}^{n} |y_i - \beta^T x_i|$$

$$\sum_{i=1}^{n} |y_i - \beta^T x_i| = \sum_{i=1}^{n} w_i(\beta)(y_i - \beta^T x_i)^2$$

$$w_i(\beta) = \frac{1}{|y_i - \beta^T x_i|}$$

If «β is known» we can do weighted least squares regression

* Start with $w_i^{(0)} = 1$. (= least squares regression) to get $\beta^{(0)}$

* In iteration k set $w_i^{(k)} = \frac{1}{|y_i - \beta^{(k-1)T} x_i|}$ or $\min\left\{\frac{1}{|y_i - \beta^{(k-1)T} x_i|}, W_{\max}\right\}$

IRLS.R

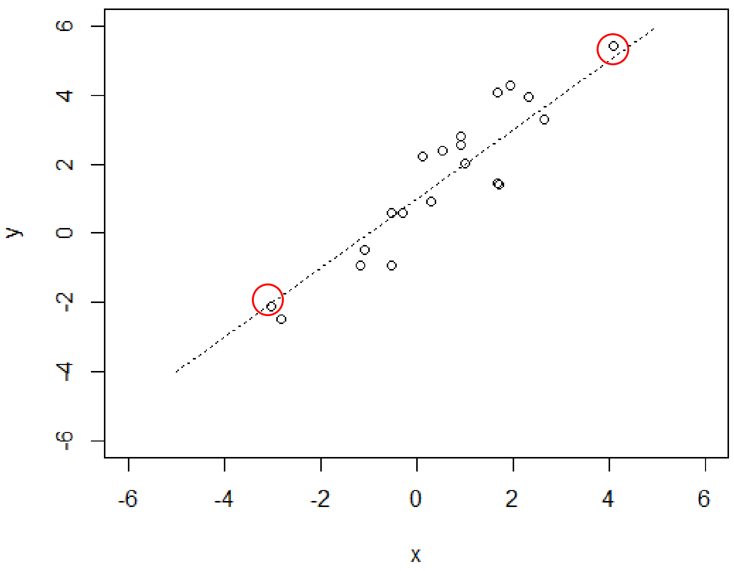
```
IRLS_L1 = function(X,y)
{

  betaHat = solve(t(X)%*%X) %*%(t(X)%*%y )
  pred0 = X%*%betaHat
  res0  =(y-pred)

  betaPrev=c(0,0)
  betaWHat=betaHat
  it=0

  while(sum(abs(betaPrev-betaWHat))>0.0001 & it<100)
  {   maxW=10
      it=it+1
      betaPrev=betaWHat
      pred= Xdata%*%betaPrev
      res=(y-pred)

      w = 1/abs(res)
      w[w>maxW]=maxW  # adjustment to avoid super large numbers [ size relative to problem]
      W = diag(as.vector(w))

      betaWHat = solve(t(Xdata)%*%W%*%Xdata) %*%(t(Xdata)%*%W%*%y )
      #show(betaWHat)
  }
  betaWHat
}
```

$$w_i^{(k)} = \min\left\{\frac{1}{|y_i - \beta^{(k-1)T} x_i|}, W_{\max}\right\}$$

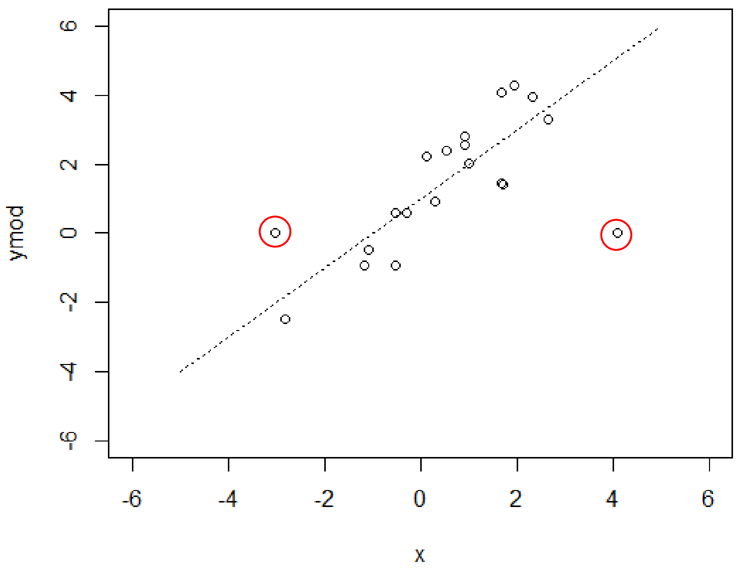
$$\beta = (\boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{y}$$

Example n=20

Case 1



Case 2



Example n=20

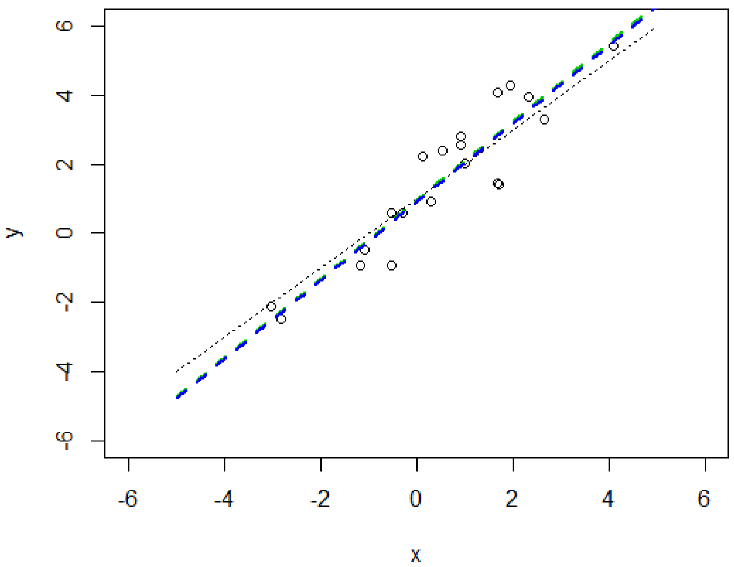
Quadratic loss - - - - -

Absolute loss - - - - -

Case 1

```
> betaHat_L2
       [,1]
  0.9504518
x 1.1409776
```

```
> betaHat_L1
       [,1]
  0.8972019
x 1.1343827
```

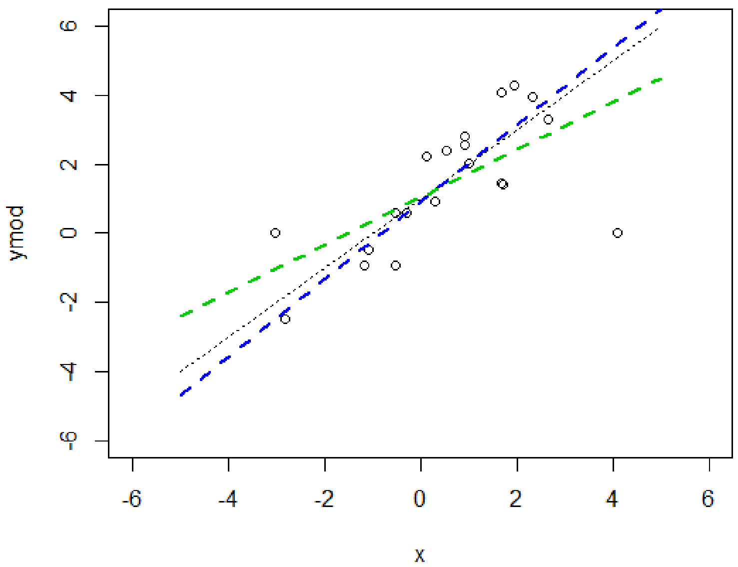


Results almost identical

Case 2

```
> betaHat_L2m
       [,1]
  1.0240550
x 0.6887221
```

```
> betaHat_L1m
       [,1]
  0.9009297
x 1.1211309
```



Quadratic loss is sensitive to outliers

P vs NP

- **P** (**P**olynomial-time)**:** decision problems that can be solved in polynomial time
- **NP** (**N**on-deterministic **P**olynomial-time)**:** decision problems that can be checked in polynomial time
- **NP-hard:** (**N**on-deterministic **P**olynomial-time hard) problems that are "at least as hard as the hardest problems in NP“ Solution to a NP-hard problem can be used to solve any NP problem
- **NP-complete:** subclass which are both NP and NP-hard. The hardest problems among NP.

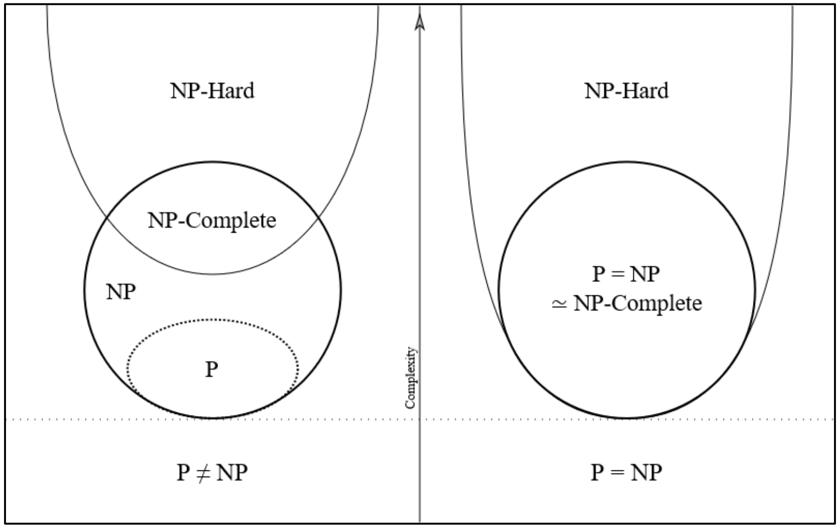
Most people think this →

Which universe do we live in?

$P \neq NP$

or

$P = NP$

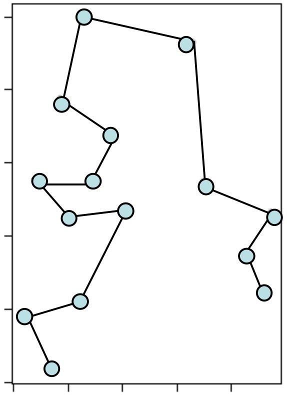


Questions:

- Can you explain the P and NP-complete and NP-hard concepts again? [see example of traveling salesperson]
- What does it actually mean to be able to check vs solve a given solution? How do you «check» a problem without solving it?
  - If someone propose a solution you can evaluate the function $f(\theta)$ -> check it. [is it better than what we have?]
  - It is harder to find a solution $\text{argmax}\, f(\theta)$ -> solve it [find the optimum value]
-

Example – Traveling salesperson problem

- A salesman needs to visit *p* cities
- Each city visited only once
- What is the minimum distance needed in order to visit all the cities?



- Example: Traveling salesperson

  Check:

  Compute the travel time along one specific path

  N – operations (N = number of cities)

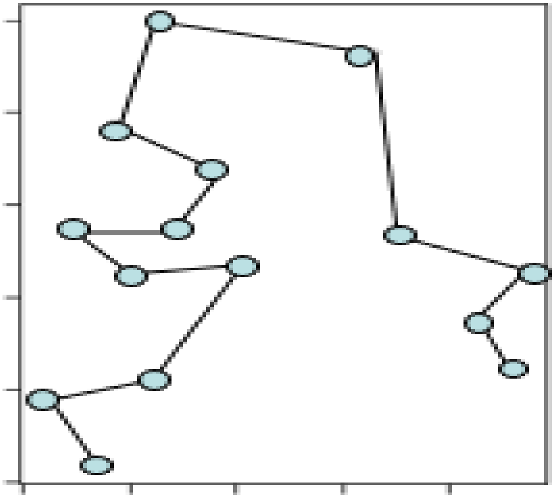
  Solve:

  Find the optimal route for the traveling salesman

  N! possibilities (number of orderings)

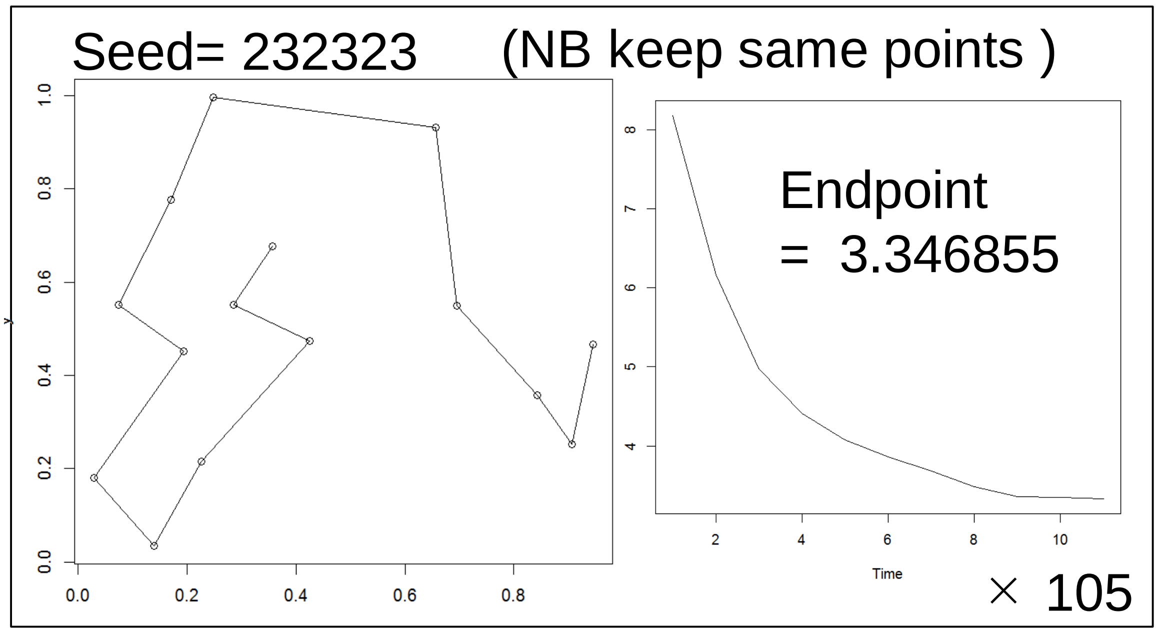
R-Examples

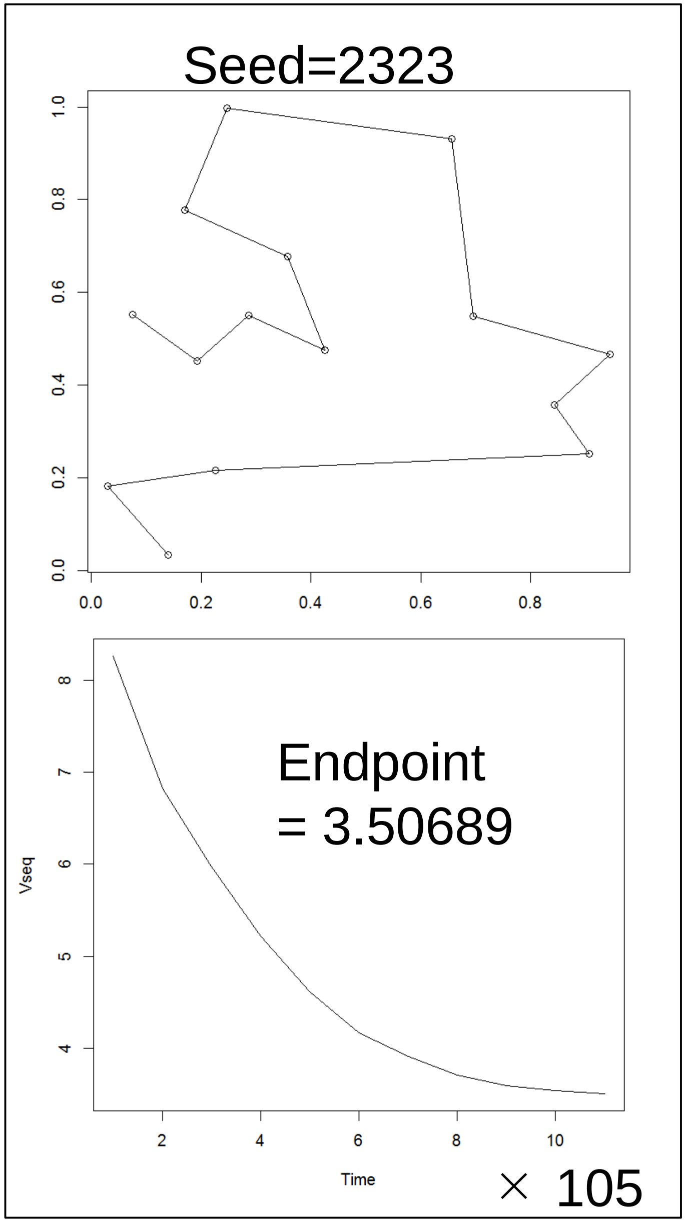
- Travelling salesman
  - Greedy
  - Simulated annealing
  - Tabu
- Baseball model selection
  - Genetic



Greedy TS

- Random initialization
- Compare all pairs of swap ($\frac{p\cdot(p-1)}{2}$)
- Select the best
- Continue until no improvement





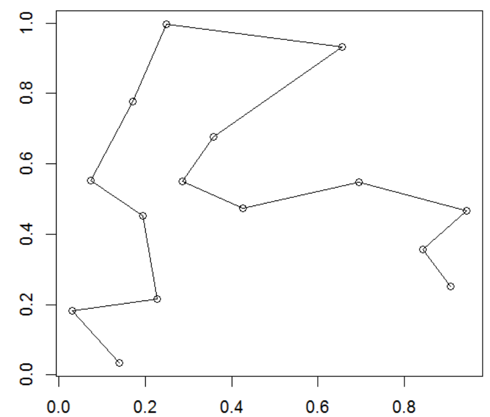
Greedy TS (Local search)

```
#Simulate positions for n=15 cities
set.seed(2323)
p = 15
pos = data.frame(x=runif(p),y=runif(p))

par(mfrow=c(1,1))
plot(pos)
#dev.copy2pdf(file="../doc/example_trav_sale.pdf")

#par(mfrow=c(3,1))
plot(pos)

#Calculate pairwise distances between cities
d = as.matrix((dist(pos,diag=TRUE,upper=TRUE)))
#image(d)

#Convert to vector in order to access many componen
d = as.vector(d)

#Random order of visits
theta = sample(1:p,p)
#Convert sequential pairs into index in the d-vect
ind = (theta[-p]-1)*p+theta[-1]   # lookup in dista
#Calculate total distance of order
V = sum(d[ind])
Vseq = V
```

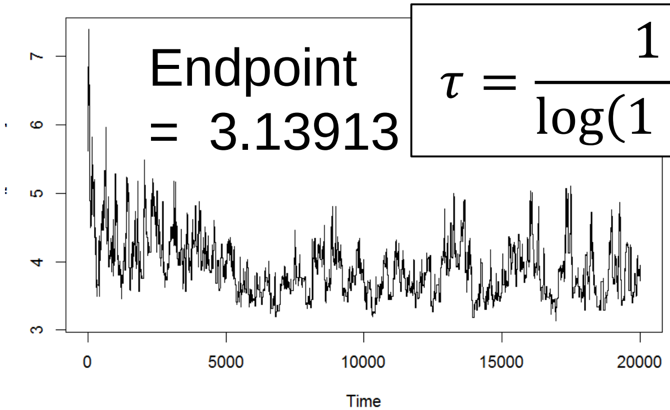
```
#Perform neighbor search, changing best two components
more = TRUE
while(more)
{
  V2opt = V
  i1opt = NA
  # loop below determines the best pair to swap
  for(i1 in 1:(p-1))
    for(i2 in (i1+1):p)
    {
      theta2 = theta
      theta2[i1] = theta[i2]
      theta2[i2] = theta[i1]
      ind2 = (theta2[-p]-1)*p+theta2[-1]
      V2 = sum(d[ind2])
      if(V2<V2opt)
      {
        V2opt = V2
        i1opt = i1
        i2opt = i2
      }
    }
    more = FALSE
    if(V2opt<V) ## if the best swap is better than current optimum  update and continiue
    {
      theta2 = theta
      theta2[i1opt] = theta[i2opt]
      theta2[i2opt] = theta[i1opt]
      theta = theta2
      V = V2opt
      Vseq = c(Vseq,V)
      more = TRUE
  }
}
```

Simulated annealing TS

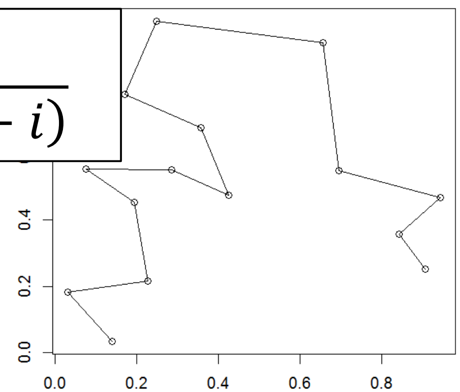
- Random initialization , evaluate $V^{(0)}$
- For each iteration draw a random pairs among ($\frac{p\cdot(p-1)}{2}$) possible
- Evaluate proposal gives value V^p
- Temperature $100/i$ or $\frac{1}{\log(1+i)}$, $m=1$
- Accept if improvement or with probability $\exp((V^{(t)}-V^p)/\tau)$
- Iterate a fixed number of times (50000) NB
  do not loop all pairs in one update

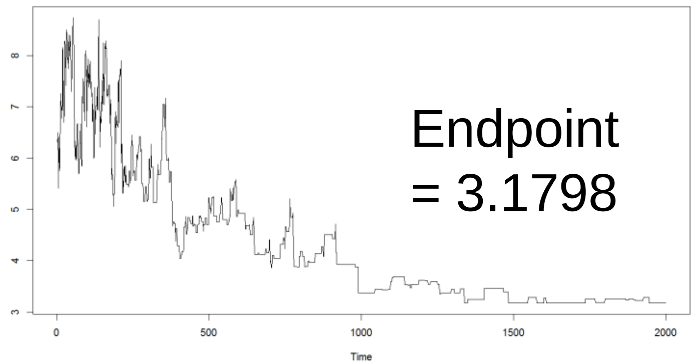


$$\tau = \frac{100}{i}$$



$$\tau = \frac{1}{\log(1+i)}$$





Simulated annealing TS

```r
Numit= 20000
for(i in 1:Numit)
{
  tau = 100/i
  #tau = 1/log(i+1)
  ind2 = sample(1:p,2,replace=F)
  theta2 = theta
  theta2[ind2[1]] = theta[ind2[2]]
  theta2[ind2[2]] = theta[ind2[1]]
  ind2 = (theta2[-p]-1)*p+theta2[-1]
  V2 = sum(d[ind2])
  prob = exp((V-V2)/tau)
  u = runif(1)
  if(u<prob)
  {
    theta = theta2
    V = V2
  }
  Vseq = c(Vseq,V)
}
```

Tabu algorithms

- Local (random) search weakness
  - Next move will in many cases reverse previous move
- Tabu idea:
  - Allow downhill move when no uphill move is possible
  - Make some moves temporarily forbidden or tabu
  - Early form: steepest ascent /mildest decent
    - Move to least unfavorable when there is no uphill move

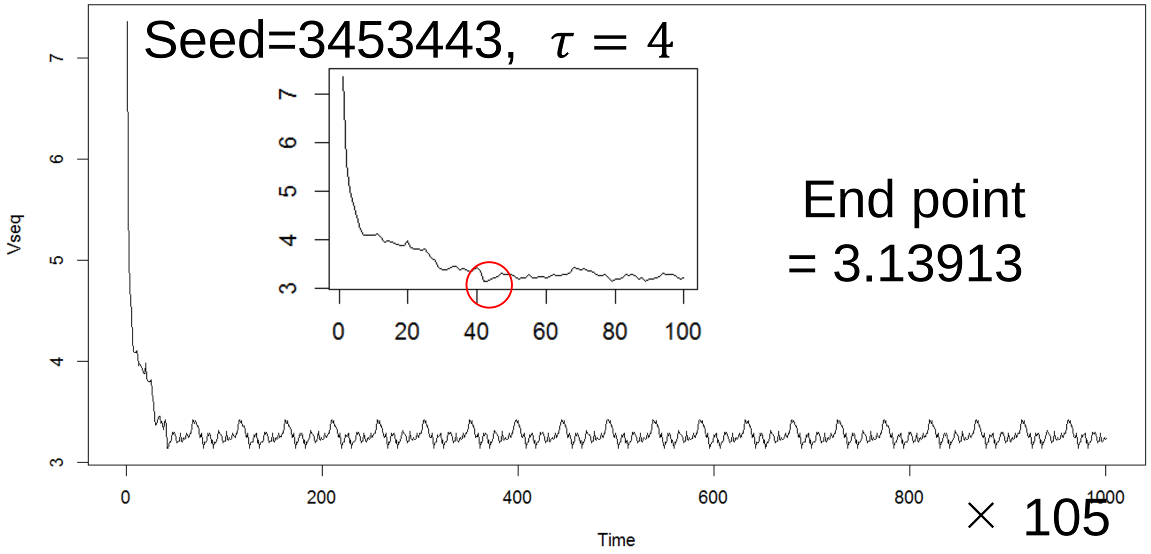
Traveling salesman Tabu

- Neighborhood: Swap the order of two components
- Move: To the best state in the neighborhood even if it is worse
- Tabu: Do not allow to pick two components that have been selected in the last k iterations
- Implementation:
  - Make a table of all possible pairs that can be picked, a $p(p-1)\times 2$ table
  - Make a list H containing the last k pairs that have been picked (references to the rows in the table above)
  - When searching within neighborhood, do not consider those pairs contained in H
  - When found the best pair, remove the first element of H and add the new pair to the end of H
- `Travel_salesman_tabu.R`

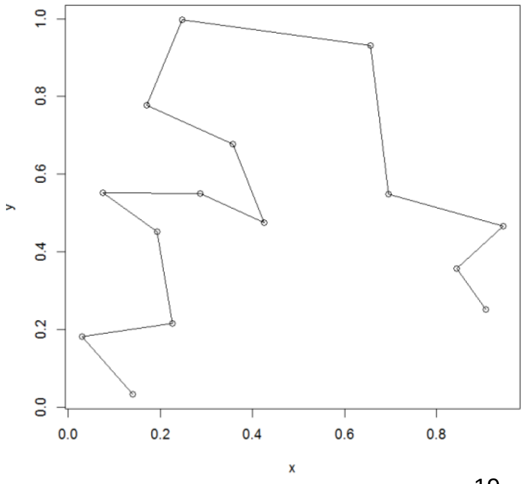
TABU TS

- Random initialization
- Compare all pairs of swap ($\frac{p\cdot(p-1)}{2}$), except the four last $\tau = 4$
- Build TABU list gradually to max size ($\tau = 4$).
- Remove FIFO when exceeding max size
- Select the best on the list
- Store the best so far
- Iterate a fixed number of times (1000)

Seed	tau	Value	First occur.
2323	4	3.5068	11
2323	10	3.1391	359
232323	4	3.2979	10
232323	10	3.1391	915
3453443	4	3.1391	42
3453443	10	3.1391	22



$\times\ 105$



```
#Perform neighbor search, changing best two components
more = TRUE
tabu = NULL
H = NULL
tau = 10
#while(more)
for(it in 1:10000)
{
  V2opt = V+1000  #Just to get some initial value to beat
  i1opt = NA
  for(i in 1:num)
  {
    if(is.na(pmatch(i,H)))
    {
      #Find indices to swap
     i1 = searchtab[i,1]
     i2 = searchtab[i,2]
     #Swap components, put into theta2
     theta2 = theta
     theta2[i1] = theta[i2]
     theta2[i2] = theta[i1]
     #Calculate value for new configuration
     ind2 = (theta2[-p]-1)*p+theta2[-1]
     V2 = sum(d[ind2])
     #If best so far, store it
     if(V2<V2opt)
     {
       V2opt = V2
       iopt = i
       i1opt = i1
       i2opt = i2
     }
    }
  }
  #Change to best configuration found
  theta2 = theta
  theta2[i1opt] = theta[i2opt]
  theta2[i2opt] = theta[i1opt]
  theta = theta2
  V = V2opt
  Vseq = c(Vseq,V)
  #Include the swap in TABU table
  H = c(H,iopt)
  #If table is too large, remove first element (oldest swap)
  if(length(H)>tau)
    H = H[-1]
  #Check if better than best so far
  if(V < Vopt)
  {
    theta.opt = theta
    Vopt = V
  }
}
```

Genetic algorithm baseball salaries

- Salaries for $n = 337$ baseball players
- $p = 27$ possible covariates, $2^{27} = 134\,217\,728$ possible models

Covariates are statistics collected during a season

- # runs scored
- batting average
- on pace percentage
- …

- Genetic algorithm (for model selection)
  - Starting with $P = 100$ models selected randomly
  - Choose two parents with probabilities proportional to $\exp(-AIC)$
  - For each component choose the state from one of the parents randomly
  - Allow mutation (change) with probability $\mu = 0.01$
  - `Baseball_genetic.R`

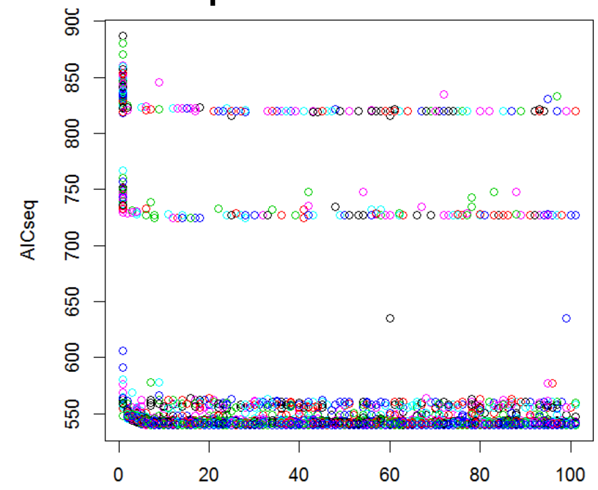
Genetic algorithm, Baseball $

- `Baseball_genetic.R`
- Maximize: $-AIC$ (model selection criteria)
- P=100 (population size)
- Chromosome length C=p=27
- Random initialization
- Select individuals with probability $\propto \exp(-AIC)$
- Mutation 1% (per locus and individual)
- 100 generations

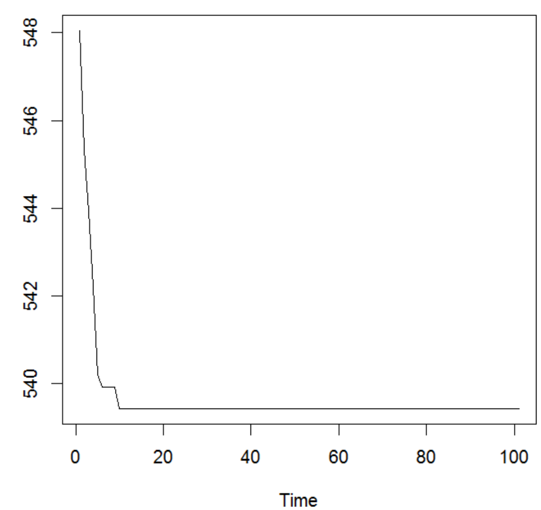
- Best achieved (first run)
  - $\theta^* =$ [2 3 6 8 10 13 14 15 16 24 25 26]
  - $f(\theta^*) = 539.4174$

Other seeds 539.4174, 541.7527

Population fit



Best in generation



```
AICseq = AICfit
more = TRUE
Numit=100
pop.new = pop
AICfit.new = AICfit
mu = 0.01   #Probability for mutation
#Start iteration on updating populations
for(i in 1:Numit)
{
 for(k in 1:P)
  {
    #Selecting parents with probability proportional to exp(-AIC)
    phi1 = exp(-AICfit)
    phi2 = exp(-AICfit)

    #Selecting parents
    parent1 = sample(1:P,1,prob=phi1)
    parent2 = sample(1:P,1,prob=phi2)

    #Sampling independently which parent to inherit from
    bred = sample(1:2,p,replace=T)
    pop.new[k,bred==1] = pop[parent1,bred==1]
    pop.new[k,bred==2] = pop[parent2,bred==2]

    #Mutation
    ind2 = sample(0:1,p,replace=T,prob=c(1-mu,mu))
    if(sum(ind2)>0)
      pop.new[k,ind2==1] = 1-pop.new[k,ind2==1]

    #Extract only those components that are selected
    ind = c(1:p)[pop.new[k,]==1]
    base2 = baseball[,c(1,1+ind)]
    #Fit the new model
    AICfit.new[k] = AIC(lm(log(salary)~.,data=base2))
  }
  pop = pop.new
  AICfit = AICfit.new
  AICseq = rbind(AICseq,AICfit)
}
```