Optimisation Prac 2

Andomei Smit: SMTAND051

25/02/2025

Contents

Simulated Annealing

Changing the solution to the travelling salesman problem

In this case we will change the way that the moves are made to change the order of the cities

```
# load data
y <- as.matrix(eurodist)
```

```
# function to initialise cities
get_initial_x <- function(ncity){
  tour <- sample(1:ncity)
  # ensure tour returns to first city
  tour <- c(tour, tour[1])

  return(tour)
}

# get initial tour:
cur_tour <- get_initial_x(nrow(y))
```

```
# define an evaluation function
evaluate_x <- function(dists, tour){
  sum(dists[cbind(tour[-length(tour)], tour[-1])])
}
```

We will change the function that perturbs by setting it to choose 2 random indeces and swopping the cities related to those cities.

```
perturb_x <- function(tour){

  # select two indeces at random
  i_j <- sample(1:(length(tour)), 2, replace = F)
  # swop the cities at these indeces
  ## select city i
```

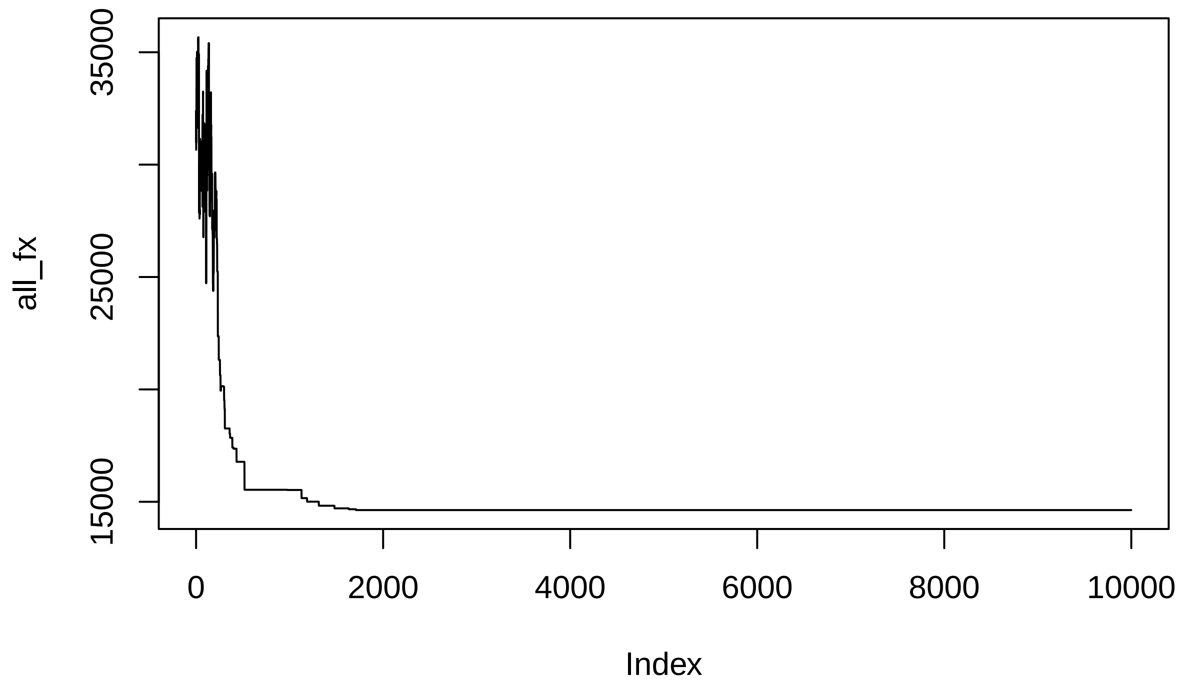
```r
  city_i <- tour[i_j[1]]
  ## select city j
  city_j <- tour[i_j[2]]
  ## swop them
  tour[i_j[1]] <- city_j
  tour[i_j[2]] <- city_i

  return(tour)
}
```

```r
# set start temperature and geometric cooling factor
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.98
```

```r
# get an initial solution
cur_x <- get_initial_x(ncol(y))
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
```

```r
# initialize results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:10000){

  # generate a candidate solution
  prop_x <- perturb_x(cur_x)

  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = prop_x)

  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)

  # accept or reject the candidate
  if(runif(1) < accept_prob){
   cur_x <- prop_x
   cur_fx <- prop_fx
  }

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- rbind(all_x,cur_x)

}
```

```r
plot(all_fx,type="l")
```



The best tour returned by the simulated annealing algorithm is given below.

```
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
best_solution
```

```
## [1] 1710
```

```
best_tour
```

```
##      X10L X1L X16L X14L X12L X7L X20L X15L X4L X19L X18L X2L X17L X9L X8L X3L
## 1710   11   3    4    5   12   9   14    2  15   19    1  21   17  16  10  10
##      X21L X6L X11L X13L X5L X10L.1
## 1710    6  18   13    8   7     20
```

```
colnames(y)[as.numeric(best_tour)]
```

```
##  [1] "Hook of Holland" "Brussels"        "Calais"          "Cherbourg"
##  [5] "Lisbon"          "Gibraltar"       "Madrid"          "Barcelona"
##  [9] "Marseilles"      "Rome"            "Athens"          "Vienna"
## [13] "Munich"          "Milan"           "Hamburg"         "Hamburg"
## [17] "Cologne"         "Paris"           "Lyons"           "Geneva"
## [21] "Copenhagen"      "Stockholm"
```

Changing the solution to the Soduko problem

```
# create a Sudoku puzzle
s <- matrix(0,ncol=9,nrow=9)
s[1,c(6,8)] <- c(6,4)
s[2,c(1:3,8)] <- c(2,7,9,5)
s[3,c(2,4,9)] <- c(5,8,2)
s[4,3:4] <- c(2,6)
s[6,c(3,5,7:9)] <- c(1,9,6,7,3)
s[7,c(1,3:4,7)] <- c(8,5,2,4)
```

```r
s[8,c(1,8:9)] <- c(3,8,5)
s[9,c(1,7,9)] <- c(6,9,1)

s
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,]    0    0    0    0    0    6    0    4    0
##  [2,]    2    7    9    0    0    0    0    5    0
##  [3,]    0    5    0    8    0    0    0    0    2
##  [4,]    0    0    2    6    0    0    0    0    0
##  [5,]    0    0    0    0    0    0    0    0    0
##  [6,]    0    0    1    0    9    0    6    7    3
##  [7,]    8    0    5    2    0    0    4    0    0
##  [8,]    3    0    0    0    0    0    0    8    5
##  [9,]    6    0    0    0    0    0    9    0    1
```

```r
# define the spaces that can be changed
free_spaces <- (s == 0)
free_spaces
```

```
##        [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]
##  [1,]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE  TRUE
##  [2,] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
##  [3,]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
##  [4,]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [5,]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
##  [6,]  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
##  [7,] FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE  TRUE  TRUE
##  [8,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
##  [9,] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

```r
get_initial_x <- function(s){

  # identify free spaces
  free_spaces <- (s == 0)
  # fixed non-free spaces
  cur_x <- s
  # randomly choose a number between 1 and 9 for free spaces
  cur_x[free_spaces] <- sample(1:9, sum(free_spaces), replace=T)

  return(cur_x)
}

# create the initial solution:
init_x <- get_initial_x(s)
```

These functions will stay the same.

```r
# function to count duplicates:
count_duplicates <- function(x)
{
  n_dup <- length(x) - length(unique(x))
  return(n_dup)
}

# sum duplicates accross rows, columns and blocks:
```

```r
evaluate_x <- function(x){

  # within-row duplications
  row_dups <- sum(apply(x,1,count_duplicates))

  # within-col duplications
  col_dups <- sum(apply(x,2,count_duplicates))

  # within-block duplications
  block_dups <- 0
  for(i in 1:3){
    for(j in 1:3){
      small_x <- x[(3*(i-1) + 1):(3*i), (3*(j-1) + 1):(3*j)]
      thisblock_dups <- count_duplicates(as.vector(small_x))
      block_dups <- block_dups + thisblock_dups
    }
  }

  total_dups <- row_dups + col_dups + block_dups

  return(total_dups)

}

# evaluate performance of the initial solution:
evaluate_x(init_x)
```

```
## [1] 76
```

We will change the perturbing in this problem to replace one of the free_spaces values in the column and row that has the most duplicates with a random draw. The idea is that if we target the most problematic columns first, the algorithm should run faster.

```r
perturb_x <- function(x, free_spaces)
{
  # find column with the largest number of duplicates
  max_col_index <- which.max(apply(init_x, 2, count_duplicates))

  # select a free site
  row_index <- sample(1:9,1,prob=free_spaces[,max_col_index])
  # change that site at random
  x[row_index, max_col_index] <- sample(1:9,1,replace=T)

   # find row with the largest number of duplicates
  max_row_index <- which.max(apply(init_x, 1, count_duplicates))

  # select a free site
  col_index <- sample(1:9,1,prob=free_spaces[max_row_index,])

  # change that site at random
  x[max_row_index, col_index] <- sample(1:9,1,replace=T)

  return(x)
}
```

We now apply simulated annealing to solve the Sudoku puzzle, starting by setting a start temperature and geometric cooling factor.

```r
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995
```

We'll generate a new initial solution and evaluate, just to keep this whole code block self-contained.

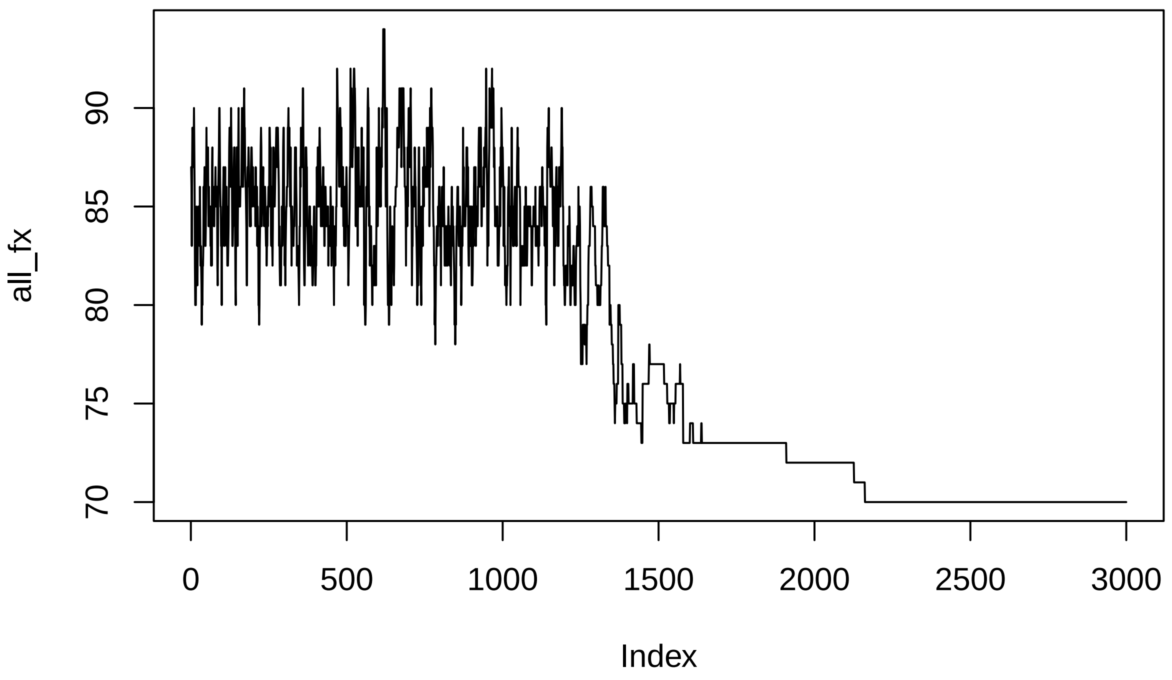
```r
cur_x <- get_initial_x(s)
cur_fx <- evaluate_x(cur_x)
```

We now apply the simulated annealing algorithm.

```r
# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:3000){

  # generate a candidate solution
  prop_x <- perturb_x(cur_x, free_spaces = free_spaces)

  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)

  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)

  # accept or reject the candidate
  if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- rbind(all_x,as.vector(cur_x))

}
```

We plot the solution quality over time.

```r
plot(all_fx,type="l")
```



From the above plot we can see that this peturbing function actually performs worse than the previous model.

Genetic Algorithms

Changing the solution to the traveling salesman problem

```r
# Generate initial population of solutions (given number of cities and size of popn)
initial_pop = function(varsize, popsize){
  # sample 1:21 popsize number of times
  pop0 <- matrix(NA, popsize, varsize)
  for(i in 1:popsize){
    pop0[i,] <- sample(1:nrow(y), replace = F)
  }
  return(pop0)
}
```

```r
# evaluate the total distance for each item in the population
evaluate_pop = function(pop, y){
  # y is the distance matrix

  # create empty eval vector:
  evals = rep(0,nrow(pop))
  for(i in 1:nrow(pop)){
    # sum pairwise distances between city for each population member
    evals[i] <- sum(y[cbind(pop[i,][-length(pop[i,])], pop[i,][-1])])
  }
  return(evals)
}
```

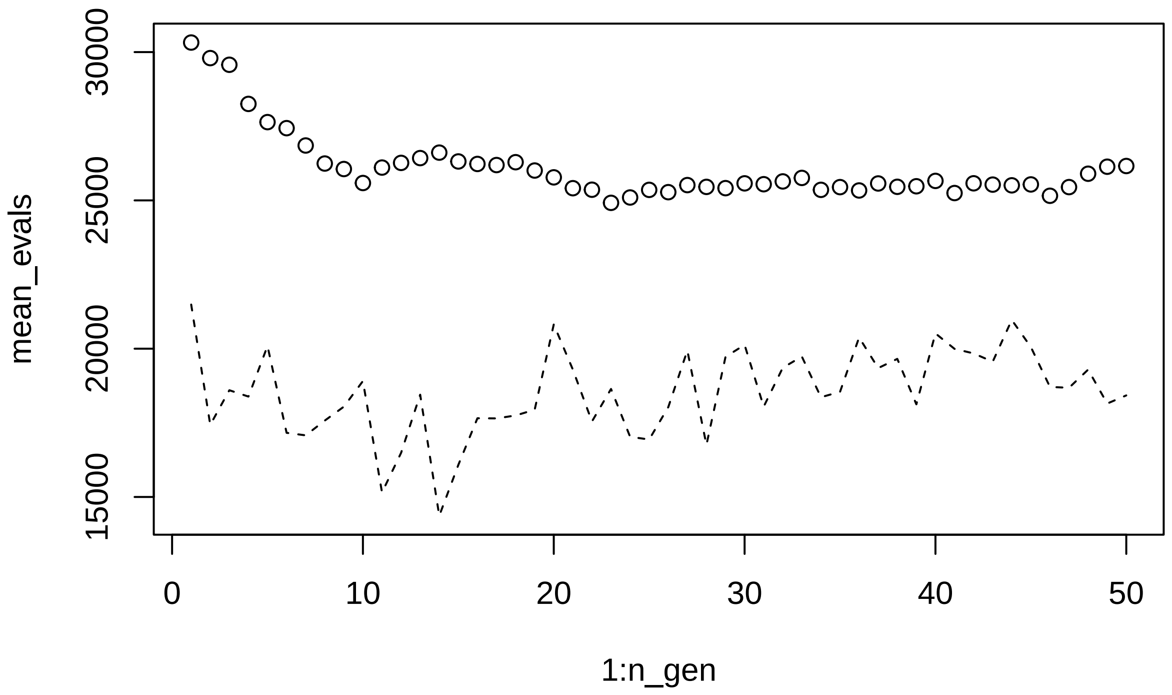
```r
# Function for selection (outputs = parents)
# proportional to fitness
select = function(pop,fitness){
  newpop = matrix(NA, nrow(pop), ncol(pop))
  # randomly select how many you want to change
  # note: change the prob to be inverted (smaller distances have greater prob)
  probabilities <- dnorm(scale(fitness))
  picks = sample(1:nrow(pop), prob = probabilities, replace = T)
  for(i in 1:length(picks)){
    # replace the columns newpop with the random draw of the old population
    newpop[i, ] = pop[picks[i], ]
  }
  return(newpop)
}
```

```r
# Function for crossover step (outputs = offspring_crossover)
#uniform crossover
crossover = function(parents){
  popsize = nrow(parents)
  varsize = ncol(parents)
  # Pick parents to mate randomly
  parent_pairs = matrix(sample(1:popsize), popsize/2, 2)
  # Initialise offspring
  offsprings = matrix(NA, popsize, varsize)
  for(i in 1:nrow(parent_pairs)){
    # Get parents
    p1 = parents[parent_pairs[i,1], ]
    p2 = parents[parent_pairs[i,2], ]
    # Make kids
    c1 = rep(NA, varsize)
    c2 = rep(NA, varsize)
    # Apply uniform crossover to get kids
    for(j in 1:varsize){
      if(runif(1) <= 0.5){
        c1[j] = p1[j] # c1's jth element is same as p1's jth element
        c2[j] = p2[j]
      }else{
        c2[j] = p1[j]
        c1[j] = p2[j]
      }
    }
    # Store kids
    offsprings[2*i-1, ] = c1
    offsprings[2*i, ]   = c2
  }
  return(offsprings)
}
```

```r
# Function for mutation step (outputs = offspring_mutation) (no changes made)
#scramble mutation
mutation = function(offspring_crossover,mutation_rate=0.05){
  # Initialise mutations
  mutations = matrix(NA, nrow(offspring_crossover), ncol(offspring_crossover))

  for(i in 1:nrow(offspring_crossover)){
    persontomutate = offspring_crossover[i,]
    if(runif(1) <= mutation_rate){
      # Select two elements
      picks = sort(sample(1:ncol(offspring_crossover),
                          2, replace = FALSE))
      # Get sub-set
      temp = persontomutate[picks[1]:picks[2]]
      # Reshuffle
      temp = sample(temp, length(temp), replace = FALSE)
      # Add mutation
      persontomutate[picks[1]:picks[2]] = temp
      mutations[i,] = persontomutate
    }else{
      mutations[i,] = persontomutate
    }
  }
  return(mutations)
}
```

```r
# Function for replacement (outputs = new population)
#generational replacement
replace = function(parents,offspring_mutation){
  return(offspring_mutation)
}
```

```r
## putting it all together
#pts = c(10, 20, 15, 2, 30,10, 30)
#wts = c(1, 5, 10, 1, 7, 5, 1)
set.seed(10)
n_pop = 100
n_vars = ncol(y)
this_pop = initial_pop(varsize=n_vars,popsize=n_pop)

min_evals = c()
mean_evals = c()
n_gen = 50

for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, y=y)
  next_parents = select(pop=this_pop,fitness=evals)
  offspring_crossover = crossover(parents=next_parents)
  offspring_mutation = mutation(offspring_crossover,mutation_rate=0.05)
  this_pop = replace(next_parents, offspring_mutation)
  min_evals = c(min_evals,min(evals))
  mean_evals = c(mean_evals,mean(evals))
}

plot(1:n_gen,mean_evals,
     ylim=c(min(c(min_evals,mean_evals)),max(c(min_evals,mean_evals))))
lines(1:n_gen,min_evals,lty=2)
```


mean_evals
1:n_gen
15000
20000
25000
30000
0
10
20
30
40
50