Inteligencja Obliczeniowa

Laboratorium 3-5

Autorzy:
Maciej Kiedrowski, nr indeksu: 200105
Wojciech Then, nr indeksu: 196057

Grupa: Środa 18:55

Data oddania: 13.05.2017

Prowadzący: Dr hab. inż. Olgierd Unold

Spis treści

1 Własne funkcje krzyżowania i mutacji

1.1 Krzyżowanie

1.1.1 Funkcja testowa

Testu algorytmu zostały wykonane dla funkcji wielomodalnej nr. 13 z biblioteki *cec2013*.

13) Non-continuous Rotated Rastrigin's Function

$$f_{13}(x) = \sum_{i=1}^{D} (z_i^2 - 10\cos(2\pi z_i) + 10) + f_{13}^* \qquad (13)$$

$$\hat{x} = \mathbf{M}_1 \frac{5.12(x-o)}{100}, y_i = \begin{cases} \hat{x}_i & \text{if } |\hat{x}_i| \le 0.5 \\ round(2\hat{x}_i)/2 & \text{if } |\hat{x}_i| > 0.5 \end{cases} \text{ for } i = 1,2,\ldots,D$$

$$z = \mathbf{M}_1 \Lambda^{10} \mathbf{M}_2 T_{asy}^{0.2}(T_{osz}(y))$$

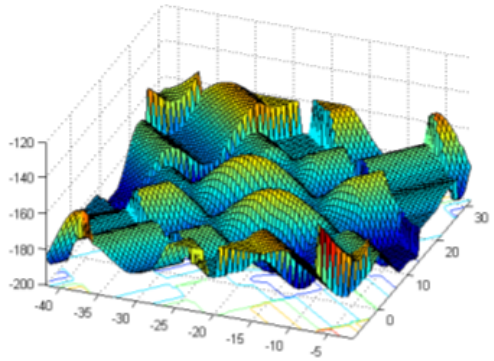


Figure 13(a). 3-*D* map for 2-*D* function

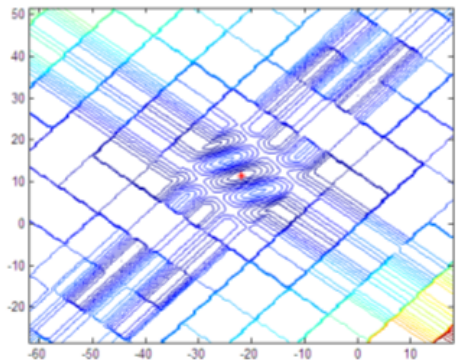


Figure 13(b).Contour map for 2-*D* function

Rysunek 1: Funkcja testowa

1.1.2 Założenia testów

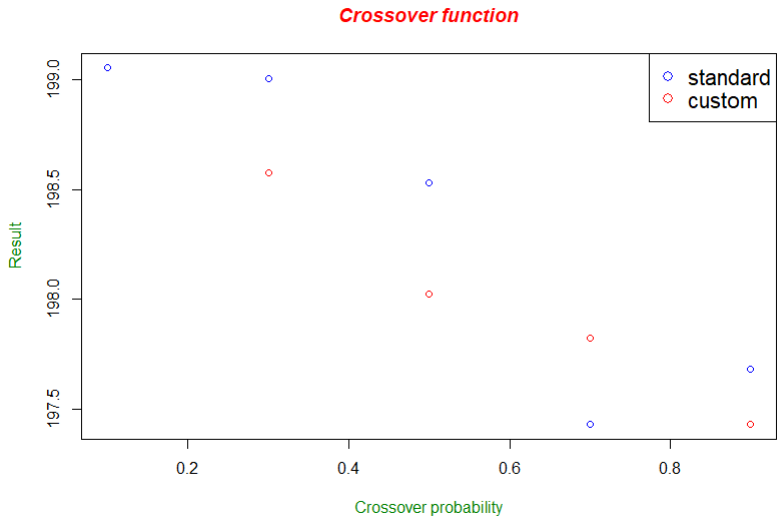
W celu przetestowania możliwości użycia własnej funkcji krzyżowania zmodyfikowana została standardowa funkcja *gareal_waCrossover* z pakietu *GA*. Oryginalny współczynnik służący do określenia proporcji parametrów rodziców w potomstwie, określony przez rozkład jednostajny na zakresie (0-1) zastąpiony został wartością stałą, wynoszącą odpowiednio 0,6 i 0,4 dla rodziców.
Wykonane zostały badania przy różnych wartościach parametru odpowiadającego za szansę krzyżowania oraz wielkości populacji. Do analizy wyników posłużyły wartość znalezionego minimum oraz liczba iteracji, po których algorytm kończył działanie. Pozwala to analizować jednocześnie jakość wyniku i czas potrzebny na jego znalezienie - co w przypadku rzeczywistych zastosowań ma równie duże znaczenie.
Wszystkie przedstawione wyniki są uśrednieniem wyników z 30 uruchomień algorytmu, maksymalna ilość iteracji to 250, obszar poszukiwań minimum:

$$x \in [-60, 10]$$

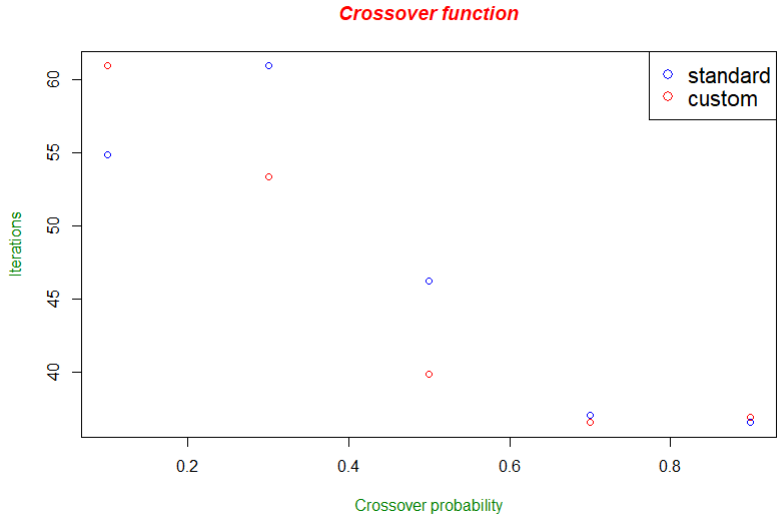
$$y \in [-50, 20]$$

Pozostałe parametry przyjmowały wartości domyślne dla pakietu *GA*.

1.1.3 Wyniki w zależności od prawdopodobieństwa krzyżowania

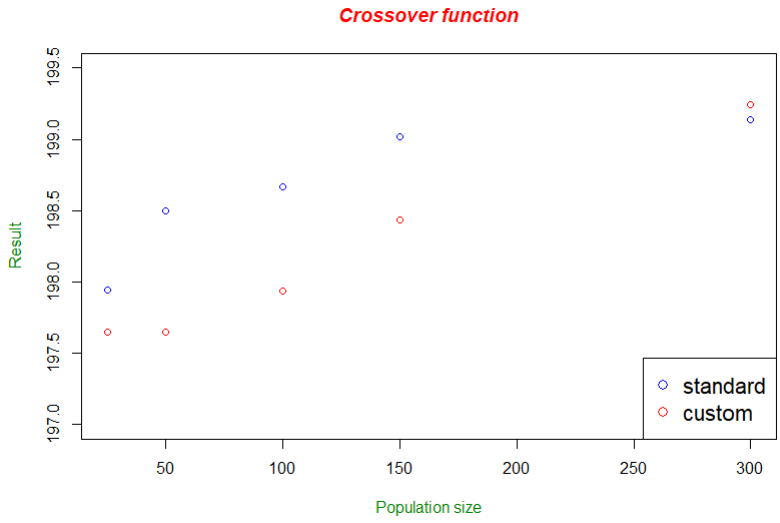


Rysunek 2: Rezultat optymalizacji dla różnych wartości prawdopodobieństwa krzyżowania

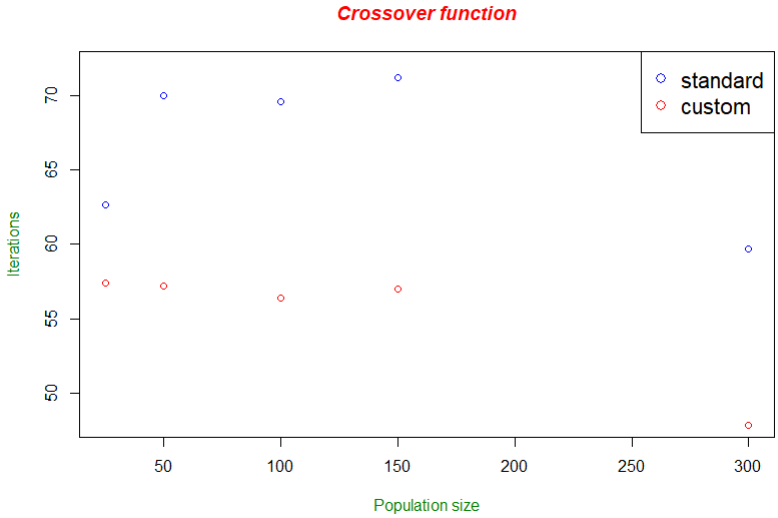


Rysunek 3: Ilość iteracji dla różnych wartości prawdopodobieństwa krzyżowania

1.1.4 Wyniki w zależności od wielkości populacji



Rysunek 4: Rezultat optymalizacji dla różnych wielkości populacji



Rysunek 5: Ilość iteracji dla różnych wielkości populacji

1.1.5 Wnioski

Modyfikacja algorytmu poprzez wyeliminowanie zmiennej losowej z funkcji krzyżowania spowodowała pogorszenie osiąganych przez algorytm genetyczny wyników. Algorytm genetyczny po modyfikacji krzyżowania słabiej odnajduje minimum w ramach pojedynczego minimum lokalnego co przedstawia się w mniejszej ilości iteracji wykonywanych przez algorytm - przy braku postępu funkcja stopu kończy działanie programu.

1.2 Mutacja

1.2.1 Funkcja testowa

Testy zostały przeprowadzone dla tej samej funkcji wielomodalnej, co w przypadku testów podmiany funkcji krzyżowania, czyli funkcji 13 z pakietu cec2013.

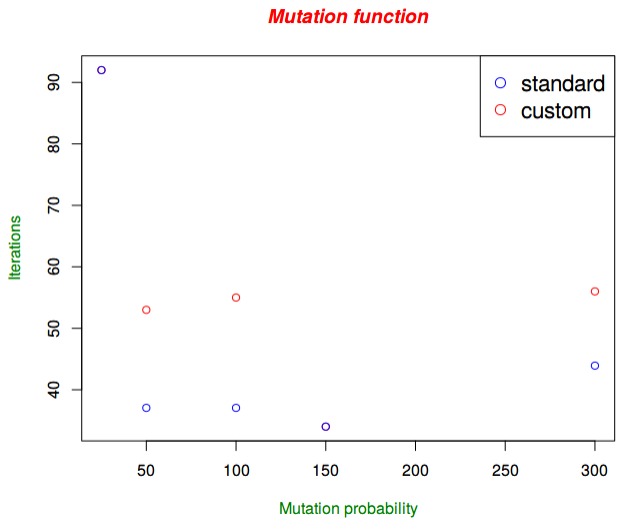
1.2.2 Założenia testów

W ramach testów zachowania się GA przy zmianie funkcji mutacji, posłużono się zmodyfikowaną funkcją *gareal_ nraMutation (non uniform random mutation)* z pakietu *GA*.
W funkcji zmieniono współczynnik tłumienia (*dempening factor*) z obliczanego na postawie bieżącej iteracji oraz maksymalnej liczby iteracji na stały o wartości 0.1.
Badania wykonano zmieniając wartości szansy mutacji oraz wielkości populacji. Reszta parametrów, jest taka sama, jak w przypadku podmiany funkcji krzyżowania:

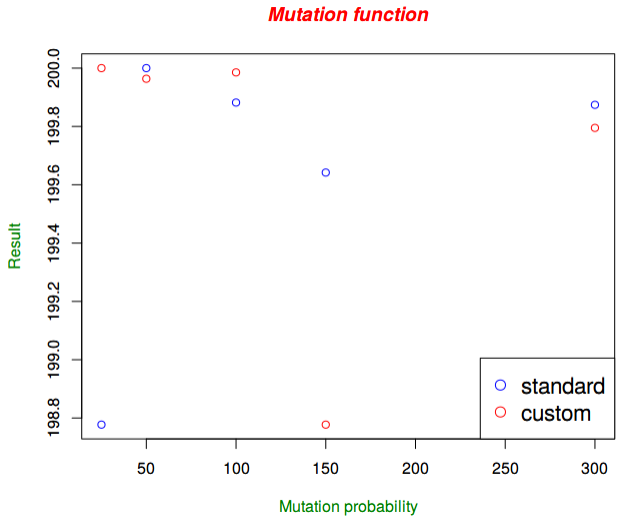
- analizę wniosków oparto o wartość znalezionego minimum i liczbę iteracji, po których algorytm kończył działanie
- wyniki są uśrednieniem 30 uruchomień algorytmu
- maksymalna ilość iteracji to 250
- obszar poszukiwania minimum:

$$x \in [-60, 10]$$
$$y \in [-50, 20]$$

1.2.3 Wyniki w zależności od prawdopodobieństwa mutacji

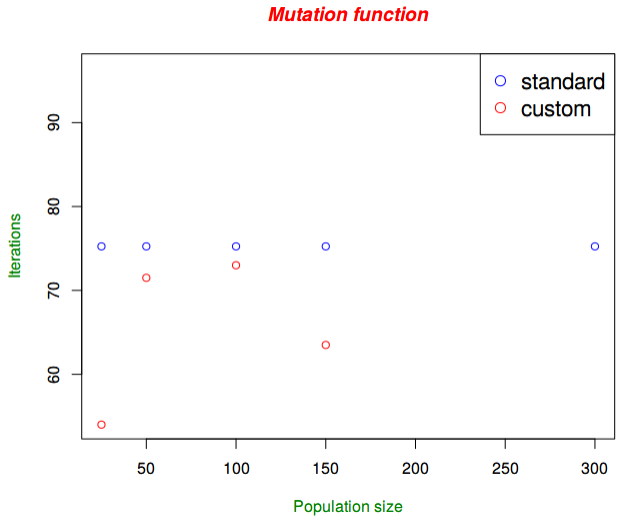


Rysunek 6: Ilość iteracji dla różnych wartości prawdopodobieństwa mutacji

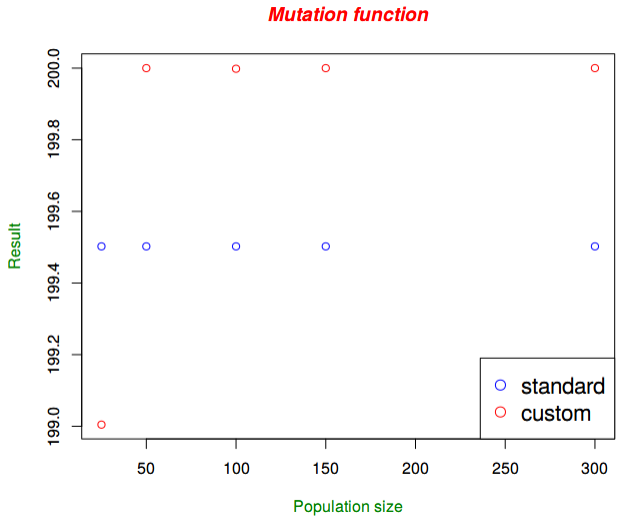


Rysunek 7: Rezultat optymalizacji dla różnych wartości prawdopodobieństwa mutacji

1.2.4 Wyniki w zależności od rozmiaru populacji



Rysunek 8: Ilość iteracji dla różnych rozmiarów populacji



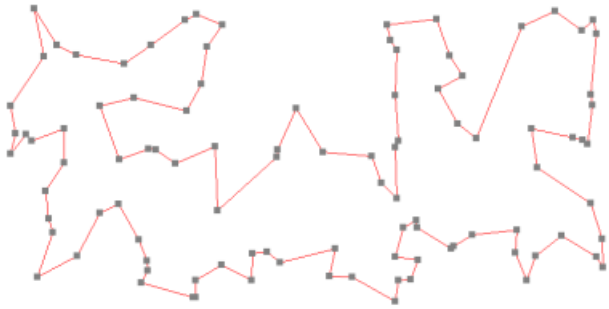
Rysunek 9: Rezultat optymalizacji dla różnych rozmiarów populacji

1.2.5 Wnioski

Modyfikacja algorytmu ponownie spowodowała znaczące pogorszenie osiąganych przez algorytm genetyczny wyników.
Algorytm genetyczny po modyfikacji mutacji nieco lepiej za to odnajduje minimum w ramach pojedynczego minimum lokalnego - ga wykonuje więcej iteracji przed sytuacją, w której postęp nie występuje. Sytuacja ta ma miejsce kiedy modyfikowane jest prawdopodobieństwo mutacji.

2 TSP

Badania dla problmu TSP zostały wykonane z użyciem instancji problemu *kroa100*, o rozwiązaniu 21282.



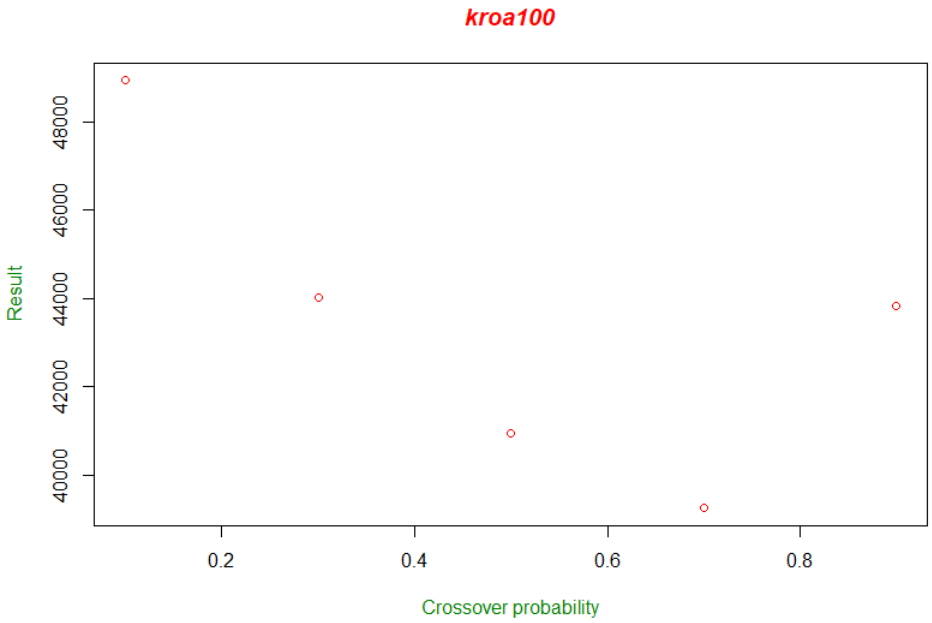
Rysunek 10: Rozwiązanie optymalne problemu kroa100

Parametry Badania wpływu parametrów na uzyskiwane wyniki zostały przeprowadzone dla parametrów określających szansę na krzyżowanie oraz wielkość populacji. Wyniki zostały uśrednione z 15 przebiegów algorytmu. Ustawienia:

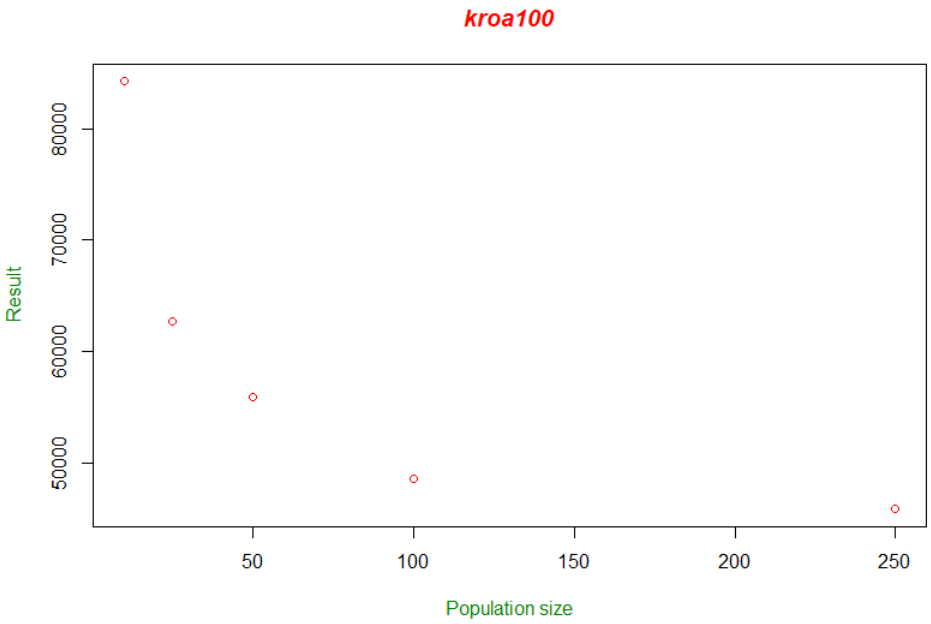
- popSize = 50
- maxIter = 500
- run = 100

Pozostałe parametry posiadały wartość domyślną.

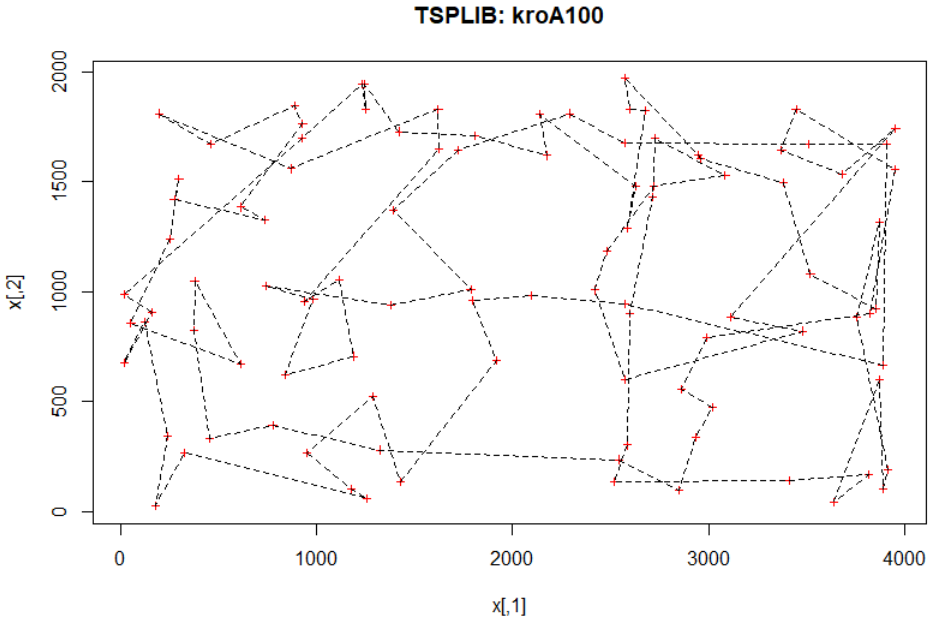
2.1 Wyniki



Rysunek 11: Rezultat optymalizacji dla różnych wartościach prawdopodobieństwa krzyżowania



Rysunek 12: Rezultat optymalizacji dla różnych wielkości populacji



Rysunek 13: Przykładowa trasa o długości 40 000

2.2 Wnioski

Zwiększenie parametru *pcrossover* powyżej wartości domyślnej 0,8 powoduje spadek wyników osiąganych przez algorytm.
Wielkość populacji ma bezpośredni wpływ na osiągane rezultaty - jej zwiększanie poprawia końcowy wynik. Zwiększanie tego parametru powoduje jednak znaczące wydłużenie czasu działania algorytmu, natomiast zysk stopniowo się zmniejsza. W czasie przeprowadzania badań najkorzystniejszą wielkością populacji okazało się 100 - powyżej tej wartości długość obliczeń dla pojedynczej iteracji jest zbyt duża. Kompromisem jest zmniejszanie populacji a zwiększanie dopuszczalnej liczby maksymalnej iteracji.
Aby określić najkorzystniejsze parametry należałoby przeprowadzić badania algorytmu dla tych parametrów, porównując osiągane wyniki po zadanym czasie, np. 1 minuty.

3 Algorytm genetyczny hybrydowy (memetyczny)

3.1 Założenia

content

3.2 Wyniki

content

3.3 Wnioski

content

4 Kody źródłowe

4.1 Custom Functions

```
# Crossover function taken untouched from
# https://github.com/cran/GA/blob/master/R/genope.R
gareal_waCrossover <- function(object, parents, ...)
{
  # Whole arithmetic crossover
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)
  children <- matrix(as.double(NA), nrow = 2, ncol = n)
  a <- runif(1)
  children[1,] <- a*parents[1,] + (1-a)*parents[2,]
  children[2,] <- a*parents[2,] + (1-a)*parents[1,]
  out <- list(children = children, fitness = rep(NA,2))
  return(out)
}

# modification of gareal_waCrossover
customCrossover <- function(object, parents, ...)
{
  # Whole arithmetic crossover
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)
  children <- matrix(as.double(NA), nrow = 2, ncol = n)
  # a <- runif(1)
  a <- 0.6
  b <- 0.4
  children[1,] <- a*parents[1,] + b*parents[2,]
  # b <- runif(1)
  children[2,] <- b*parents[2,] + a*parents[1,]
  out <- list(children = children, fitness = rep(NA,2))
  return(out)
}
```

```
# taken untouched from
# https://github.com/cran/GA/blob/master/R/genope.R
gareal_nraMutation <- function(object, parent, ...)
{
  # Non uniform random mutation
  mutate <- parent <- as.vector(object@population[parent,])
  n <- length(parent)
  g <- 1 - object@iter/object@maxiter # dempening factor
  sa <- function(x) x*(1-runif(1)^g)
  j <- sample(1:n, 1)
  u <- runif(1)
  if(u < 0.5)
```

```
  { mutate[j] <- parent[j] - sa(parent[j] - object@max[j]) }
  else
  { mutate[j] <- parent[j] + sa(object@max[j] - parent[j]) }
  return(mutate)
}

# modification of gaperm_scrMutation
customMutation <- function(object, parent, ...)
{
  # Non uniform random mutation
  mutate <- parent <- as.vector(object@population[parent,])
  n <- length(parent)
  g <- 0.1                                    # <- CHANGE
  sa <- function(x) x*(1-runif(1)^g)
  j <- sample(1:n, 1)
  u <- runif(1)
  if(u < 0.5)
  { mutate[j] <- parent[j] - sa(parent[j] - object@max[j]) }
  else
  { mutate[j] <- parent[j] + sa(object@max[j] - parent[j]) }
  return(mutate)
}
```

```
rm(list = ls())

library(GA)
library(cec2013)

source("customCrossover.R")
source("customMutation.R")
source("cecF.R")

pcross  <- c(0.1, 0.3, 0.5, 0.7, 0.9)
pmut    <- c(0.1, 0.3, 0.5, 0.7, 0.9)
popSize <- c(25, 50, 100, 150, 300)
parameterLenghts <- length(popSize)
# number of cycles of GA executions
cycles <- 2

# cec2013 function index
cecNo <- 13


# run GA with swapped mutation/crossover function
results <- matrix(0,parameterLenghts,cycles)
iterations <- matrix(0,parameterLenghts,cycles)
# meanValues <- matrix(0,parameterLenghts,cycles)
```

```
for (j in 1:parameterLenghts)
{
for(i in 1:cycles) {
  GA <- ga(
    type = "real-valued",      # for optimization problems where the decision variable
    fitness =  function(x) -cecF(x[1], x[2]),  # fitness function - takes an individ
                                               # and returns a numerical value descr
    min = c(-60, -20),         # a vector of length equal to the decision variables p
    max = c(10, 50),           # like above but maximum; c() function combines argume
    popSize = popSize[j],      # population size
    # maxiter = maxIters[j],   # maximum number of iterations to run before the GA se
    run = 30,                  # the number of consecutive generations without any im
    # elitism                  # number of best fitness individuals
    # pmutation  = pmut[j],      # probability of mutation
    # pcrossover = pcross[j]   # probability of crossover
    # crossover  = customCrossover # custom crossover function assignment
    mutation   = customMutation  # custom mutation function assignment
  )
  results[j,i] <- GA@fitnessValue
  # meanValues[i] <- tail(GA@summary[,"mean"], n=1)
  iterations[j,i] <- GA@iter
}
}


# run GA with default mutation/crossover function

resultsDef <- matrix(0,parameterLenghts,cycles)
iterationsDef <- matrix(0,parameterLenghts,cycles)
# meanValuesDef <- matrix(0,parameterLenghts,cycles)

for (j in 1:parameterLenghts)
{

for(i in 1:cycles) {
  GA_default <- ga(
    type = "real-valued",      # for optimization problems where the decision variable
    fitness = function(x) -cecF(x[1], x[2]),   # fitness function - takes an individ
                                               # and returns a numerical value descr
    min        = c(-60, -20),  # a vector of length equal to the decision variables p
    max        = c(10, 50),    # like above but maximum; c() function combines argume
    popSize    = popSize[j],   # population size
    # maxiter = maxIters[j],   # maximum number of iterations to run before the GA se
    run        = 30,           # the number of consecutive generations without any im
    # elitism                  # number of best fitness individuals
    # pmutation  = pmut[j]     # probability of mutation
    # pcrossover = pcross[j]   # probability of crossover
  )
```

```
  resultsDef[j,i] <- GA_default@fitnessValue
  # meanValuesDef[i] <- GA_default@summary[,"mean"]
  iterationsDef[j,i] <- GA_default@iter
}
}

print("Swapped:")
print(summary(GA))
plot(GA, col="red")
par(new=T)
print(mean(results))
# print(mean(meanValues))
print(mean(iterations))

print("Default:")
print(summary(GA_default))
plot(GA_default, col="green", axes=F, grid=F)
par(new=F) # dont erase previous plot (to show both)
print(mean(resultsDef))
# print(mean(meanValuesDef))
print(mean(iterationsDef))

plot(seq(1,length(GA@summary[, "max"])),GA@summary[,"max"])
```

4.2 TSP

```
rm(list = ls())

library(GA)
library(TSP)

# "global" variable used by everything
tsp <- read_TSPLIB(system.file("examples/kroA100.tsp", package = "TSP"))

# best possible
optimalTour <- solve_TSP(tsp, method = "nn", two_opt = TRUE)
plot(tsp, optimalTour, cex=.6, col = "red", pch= 3, main = "TSPLIB: kroA100")

################## implementation ################
#pcross <- c(0.1, 0.3, 0.5, 0.7, 0.9)
popSize <- c(10,25,50,100,250)
parameterLenhts <- length(popSize)
# number of cycles of GA executions
cycles <- 15


getFitness <- function(sequention, tspProblem){
```

```
  tour <- TOUR(sequention, method = NA, tsp = tspProblem)
  tour_length(tour)
}

results <- matrix(0,parameterLenhts,cycles)
iterations <- matrix(0,parameterLenhts,cycles)

for (j in 1:parameterLenhts)
{
  for(i in 1:cycles) {
    GA <- ga(
             type = "permutation",
             fitness =  function(x) -getFitness(x, tsp),
             pcrossover = pcross[j],
             min = 1,
             max = 100,
             popSize = popSize[j],
             maxiter = 250,
             run = 100
    )

    results[j,i] <- GA@fitnessValue
    iterations[j,i] <- GA@iter
  }
}

#####################################################

#possibly more than one solution - starting point may diff
result = TOUR(GA@solution[1,], method = NA, tsp = tsp)
plot(tsp, result, cex=.6, col = "red", pch= 3, main = "TSPLIB: kroA100")
```