Politechnika Wrocławska

Inteligencja Obliczeniowa i jej zastosowania

Badanie algorytmu genetycznego z zakresu optymalizacji globalnej dla wybranych funkcji testowych

Autorzy:
Paweł Andziul 200648
Marcin Słowiński 200638

Prowadzący:
dr hab. inż. Olgierd Unold,
prof. nadzw. PWr

12 kwietnia 2017

Spis treści

1 Wprowadzenie

Algorytm genetyczny – algorytm heurystyczny, który swoim działaniem przypomina działanie ewolucji w naturze. Osobniki będące zbyt słabe zostają wyeliminowane z populacji w kolejnych pokoleniach, a na ich miejsce przyjmowane są lepsze, silniejsze, bardziej podatne adaptacji. Algorytmy te zakładają możliwość mutacji i krzyżowania wśród potomków, przez co nie zawsze są oni silniejsi od poprzednio wyeliminowanych członków. Dodatkowo wprowadzają pojęcie elity, która jest bezpośrednio przenoszona do następnego - teoretycznie lepszego pokolenia.
dla wybranej funkcji wlasne funkcje krzyzowania (dla branina) dla tsp (np–trudny) genetyczny – tsplib wykorzystać do badań (2–3 instancje srednie male duze) z wlasnym operatorem z domyslnym algorytm ga z lokalnym wyszukiwaniem, dla komiwojażera, założyć czy ma lepsze wartości, czy szybciej zbiega, jak operatory się zachowują, psoptim, dla jednej funkcji i komiwojażera

W ramach laboratorium należało przeprowadzić testy algorytmu genetycznego dla różnych parametrów. Jako benchmark oceny należało użyć pakietu „getGlobalOpts" oraz języka R.

Pomiary wykonywano na 2 różnych jednostkach roboczych. Ich parametry nie są istotne z punktu widzenia analizy i możliwości porównania rezultatów.

2 Implementacja

Poniżej (listing 1) zamieszczono kod napisany w języku R przygotowany w celu umożliwienia przeprowadzenia pomiarów.

Listing 1: Skrypt w języku R wykorzystany do badań

```
1
2  # whole source code is located here:
3  # https://github.com/cran/GA/tree/master/R
4
5
6  # clean old data
7  rm(list=ls())
8  dev.off(dev.list()["RStudioGD"])
9
10 # load libraries
11 require("GA")
12 require("globalOptTests")
13 require("rgl")
14
15 # Settings ----
16
17 nOfRuns <- 2 # number of runs to calc avg scores
18
19 # colors and titles for plot series
20 colors <- c("red", "blue", "purple", "black")
21 series <- c("Seria 1", "Seria 2", "Seria 3", "Seria 4")
22
23 # default parameters for measurements
24 # each row is a different serie
25 # [mutations,crossovers,populations,iterations,color]
26 params = matrix(
```

```
  c(0, 0, 50, 100, 1,
    0, 0.8, 50, 100, 2,
    0.1, 0, 50, 100, 3,
    0.1, 0.8, 50, 100, 4),
  nrow=4, ncol=5, byrow = TRUE)

# names of functions from globalOptTests package
functions <- c("Branin")

# graph settings
graphs <- TRUE #true if you want to print graphs
quality <- 100 #number of probes

# sequences of parameters for each serie
mutationTests <- seq(0, 1, 0.1)
crossoverTests <- seq(0, 1, 0.1)
populationTests <- seq(10, 100, 5)
iterationTests <- seq(10, 200, 10)
elitismTests <- seq(0, 1, 0.1)

# Custom operators (to be done) ----

gareal_laCrossover_custom <- function(object, parents, ...)
{
  # Local arithmetic crossover
  parents <- object@population[parents,,drop = FALSE]
  n <- ncol(parents)
  children <- matrix(as.double(NA), nrow = 2, ncol = n)
  a <- runif(n)
  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)
}

gareal_sigmaSelection <- function(object, ...)
{
  # Fitness proportional selection with Goldberg's Sigma Truncation Scaling
  popSize <- object@popSize
  mf <- mean(object@fitness, na.rm = TRUE)
  sf <- sd(object@fitness, na.rm = TRUE)
  fscaled <- pmax(object@fitness - (mf - 2*sf), 0, na.rm = TRUE)
  prob <- abs(fscaled)/sum(abs(fscaled))
  sel <- sample(1:object@popSize, size = object@popSize,
                prob = pmin(pmax(0, prob), 1, na.rm = TRUE),
                replace = TRUE)
  out <- list(population = object@population[sel,,drop=FALSE],
              fitness = object@fitness[sel])
  return(out)
}

# Processing ----

customMeasure <- function(fileName, graphName, values, mType, xlab, main) {

  gMin <- .Machine$integer.max
```

```
gBest <- NA

# main measurement loop (for each serie and sequence calculate average
    results)
temp <- c()
for (defRow in 1:nrow(params)) {
  averages <- c()
  for (value in values) {
    sum <- 0
    for (i in 1:nOfRuns) {
      GAmin <- ga(type = "real-valued",
          fitness = function(xx) -f(xx),
          min = c(B[1,]), max = c(B[2,]),
          popSize = if (mType == "pop") value else params[defRow,3],
          maxiter = if (mType == "itr") value else params[defRow,4],
          pmutation = if (mType == "mut") value else params[defRow,1],
          pcrossover = if (mType == "crs") value else params[defRow,2],
          elitism = if (mType == "elt") value else max(1,
              round(params[defRow,3] * 0.05)),
          crossover = gareal_laCrossover_custom,
          optim = FALSE #hybrid ga off
          )
      solution <- matrix(unlist(GAmin@solution),ncol=dim,byrow=TRUE)
      eval <- f(solution[1,])
      if (eval < gMin) {
        gMin <- eval
        gBest <- GAmin
      }
      sum <- sum + eval
    }
    averages <- c(averages, (sum / nOfRuns))
  }
  temp <- c(temp, averages)
}
result <- matrix(c(temp),nrow = nrow(params),ncol = length(values))
write.table(result, file = paste(funcName, fileName, sep=""), row.names=FALSE,
           na="", col.names=FALSE, sep=";")

if (graphs) {

  # save graph with measurement series to file
  png(file = paste(funcName, graphName, ".png", sep=""), width=600,
      height=400, units="px")
  plot(0, 0, main=main,
       ylim=c(min(c(temp,globalOpt)),max(c(temp,globalOpt))),
       xlim=c(min(values),max(values)),
       type="n", xlab=xlab, ylab="wartosc")
  abline(globalOpt,0, col="green")
  colorNames <- c()
  seriesNames <- c()
  for (i in 1:nrow(params)) {
    color <- colors[params[i,5]]
    colorNames <- c(colorNames, color)
    seriesNames <- c(seriesNames, series[params[i,5]])
    lines(values, result[i,], col = color, type = 'l')
  }
```

```
    legend("topright", seriesNames, lwd=rep(2,nrow(params)),
        lty=rep(1,nrow(params)), col=colorNames)
    dev.off()

    summary(gBest)

    # save overview of best found minimum to file
    png(file = paste(funcName, graphName, mType, ".png", sep=""), width=600,
        height=400, units="px")
    filled.contour(x, y, z, color.palette = jet.colors, nlevels = 24,
         plot.axes = { axis(1); axis(2);
           points(solution[1,1], solution[1,2],
                  pch = 3, cex = 5, col = "black", lwd = 2)
         }
    )
    dev.off()

    # save best fitness graph to file
    png(file = paste(funcName, graphName, mType, "fitness", ".png", sep=""),
        width=600, height=400, units="px")
    plot(gBest)
    dev.off()
  }
}

for (funcName in functions) {

  # get data from globalOptTests package
   dim <- getProblemDimen(funcName)
   B <- matrix(unlist(getDefaultBounds(funcName)),ncol=dim,byrow=TRUE)
   f <- function(xx) goTest(par=c(xx, rep(0, dim-length(xx))),
                    fnName=funcName, checkDim = TRUE)
   globalOpt <- getGlobalOpt(funcName)

   if (graphs) {
     # prepare two versions of graphs (interactive and static)
     xprobes <- abs(B[2,1] - B[1,1]) / quality
     yprobes <- abs(B[2,2] - B[1,2]) / quality
     x <- seq(B[1,1], B[2,1], by = xprobes)
     y <- seq(B[1,2], B[2,2], by = yprobes)
     z <- outer(x, y, Vectorize(function(x,y) f(c(x,y))))
     png(file = paste(funcName, "1.png", sep=""), width=600, height=400,
         units="px")
     persp3D(x, y, z, theta = -45, phi = 20, color.palette = jet.colors)
     dev.off()
   }

   # for each function perform set of measurements
   customMeasure("resultsMutations.csv", "2", mutationTests, "mut",
      "p. mutacji", "Znalezione minimum dla roznych prawdopodobienstw mutacji")
   customMeasure("resultsCrossover.csv", "3", crossoverTests, "crs",
      "p. krzyzowania", "Znalezione minimum dla roznych prawdopodobienstw
          krzyzowania")
   customMeasure("resultsPopulation.csv", "4", populationTests, "pop",
      "rozmiar populacji", "Znalezione minimum dla roznych rozmiarow populacji")
   customMeasure("resultsIterations.csv", "5", iterationTests, "itr",
```

```
      "ilosc iteracji", "Znalezione minimum dla roznych ilosci iteracji")
   customMeasure("resultsElitism.csv", "6", elitismTests, "elt",
      "elityzm", "Znalezione minimum dla roznych wartosci elityzmu")
}


# library(leaflet)
#
# m <- leaflet() %>%
#   addTiles() %>% # Add default OpenStreetMap map tiles
#   addMarkers(lng=174.768, lat=-36.852, popup="The birthplace of R")
# m  # Print the map


require(GA)


# ATSP example

data("eurodist", package = "datasets")
D <- as.matrix(eurodist)
N <- max(dim(D))

tourLength <- function(tour, distMatrix) {
  tour <- c(tour, tour[1])
  route <- embed(tour, 2)[,2:1]
  sum(distMatrix[route])
}

fit <- function(tour, distMatrix) 1/tourLength(tour, distMatrix)

GA <- ga(type = "permutation", fitness = fit, distMatrix = D, min = 1, max = N,
    maxiter=2000, pmutation=0.2, run=500)

summary(GA)

apply(GA@solution, 1, tourLength, D)

mds <- cmdscale(eurodist)
x <- mds[, 1]
y <- -mds[, 2]
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "")
tour <- GA@solution[1, ]
tour <- c(tour, tour[1])
n <- length(tour)
arrows(x[tour[-n]], y[tour[-n]], x[tour[-1]], y[tour[-1]],length = 0.15, angle
    = 25, col = "steelblue", lwd = 2)
text(x, y, labels(eurodist), cex=0.8)


require(TSP)

instances <- c("u159", "u574", "u724", "u1060", "u1432", "u1817", "u2152",
    "u2319")
best_solutions <- c(42080, 36905, 41910, 224094, 152970, 57201, 64253, 234256)
found_solutions <- c()
```

```
solutions_quality <- c()

for (i in 1:length(instances)) {

  fileName = paste("examples/", instances[i], ".tsp", sep="")
  graphTitle = paste("TSPLIB: ", instances[i], sep="")

  ## Drilling problem from TSP
  drill <- read_TSPLIB(system.file(fileName, package = "TSP"))
  tour <- solve_TSP(drill, method = "nn", two_opt = TRUE)
  plot(drill, tour, cex=.6, col = "red", pch= 3, main = graphTitle)

  tl <- tour_length(tour)
  ol <- best_solutions[i]

  found_solutions <- c(found_solutions, tl)
  solutions_quality <- c(solutions_quality, ol/tl)

}

#png(file = "tsp_results.png", width=600, height=400, units="px")
plot(1:length(instances), xaxt = "n", solutions_quality, col="red",
    main="Summary", type="l", xlab="instancje", ylab="jakość rozwiązań")
axis(1, at=1:length(instances), labels=instances)
#dev.off()
```

Skrypt przygotowano w sposób który umożliwia w pełni automatyczne przeprowadzenie wszystkich pomiarów. Jednocześnie wszystkie wykresy mogą być natychmiast podmienione w sprawozdaniu. Poniżej pokrótce omówiono podstawowe parametry.

- nOfRuns

  Ilość powtórzeń dla każdego pomiaru w celu uśrednienia.

- colors, series

  Wektory kolorów i nazw kolejnych serii pomiarowych.

- params

  Macierz parametrów domyślnych algorytmu dla każdej z serii. W każdym wierszu kolejno są zawarte: p. mutacji, p. krzyżowania, rozmiar populacji, ilość iteracji oraz kolor serii na wykresach.

- functions

  Wektor nazw funkcji dla których przeprowadzane są kolejno pomiary.

Całość informacji niezbędnych do przeprowadzenia obliczeń odczytywana jest na podstawie nazwy funkcji z pakietu „globalOptTests". Są to: rozmiar problemu (ilość parametrów), domyślne ograniczenia, wartość w danym punkcie oraz optimum dla domyślnych ograniczeń.

3 Przebieg badań

Do badań zostały wybrane funkcje o różnych wymiarach zaczynając na 2 kończąc na 20. Poniżej wymieniono te funkcje wraz z ilością wymiarów podaną w nawiasie.

- Branin (2)
- Gulf (3)
- CosMix4 (4)
- EMichalewicz (5)
- Hartman6 (6)
- PriceTransistor (9)
- Schwefel (10)
- Zeldasine20 (20)

Każdy pomiar przeprowadzono 20-krotnie wyniki uśredniając co oznacza, że wartości widoczne na wykresach dla każdej serii z osobna są uśrednione po osobnych 20 przebiegach. Domyślne parametry każdej z serii przedstawiono poniżej (tabela 1). Zmianie ulegają wartości prawdopodobieństwa mutacji i krzyżowania by zbadać znaczenie ich obecności podczas optymalizacji.

Tabela 1: Parametry domyślne poszczególnych serii pomiarowych

-	Seria 1	Seria 2	Seria 3	Seria 4
Rozmiar populacji	50	50	50	50
Rozmiar iteracji	100	100	100	100
Prawdopodobieństwo mutacji	0	0	0.1	0.1
Prawdopodobieństwo krzyżowania	0	0.8	0	0.8

Zielone, poziomie linie na wykresach oznaczają optima zawarte w pakiecie „globalOptTests" dla danej funkcji przy domyślnych ograniczeniach (tych samych dla których wykonywana jest optymalizacja podczas niniejszych pomiarów).

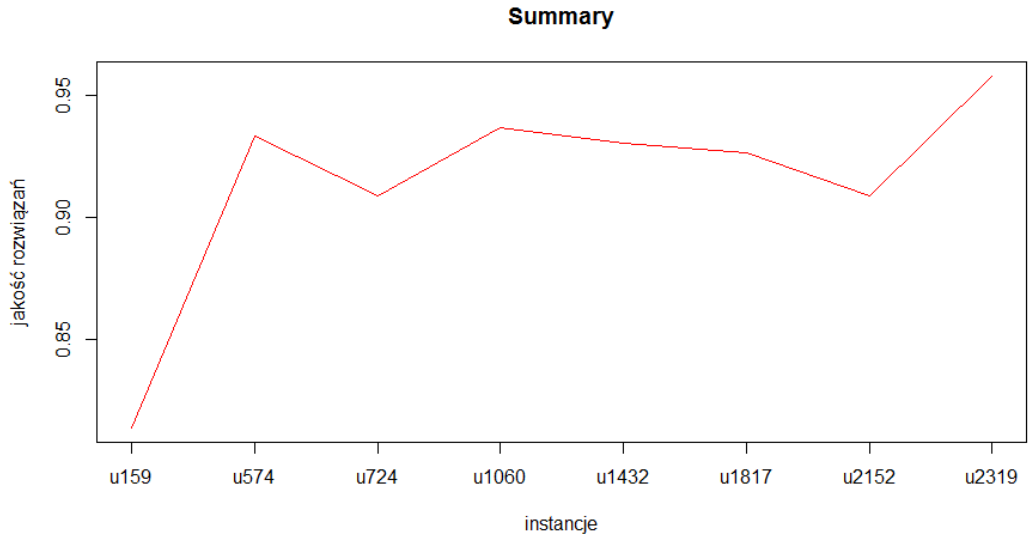
Dla funkcji o ilości parametrów większej niż 2 pominięto ilustracje graficzne znalezionych optimów gdyż optymalizacji podlegają wszystkie wymiary. Ilustracja dla dwóch pierwszych nie niesie ze sobą przydatnej informacji.

3.1 Branin (2 parametry)

Branin jest funkcją z dwoma parametrami. Na ilustracji (rys. 1) przedstawiono jej wykres a poniżej jej wzór (1) [4].

$$f(\boldsymbol{x}) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s \tag{1}$$

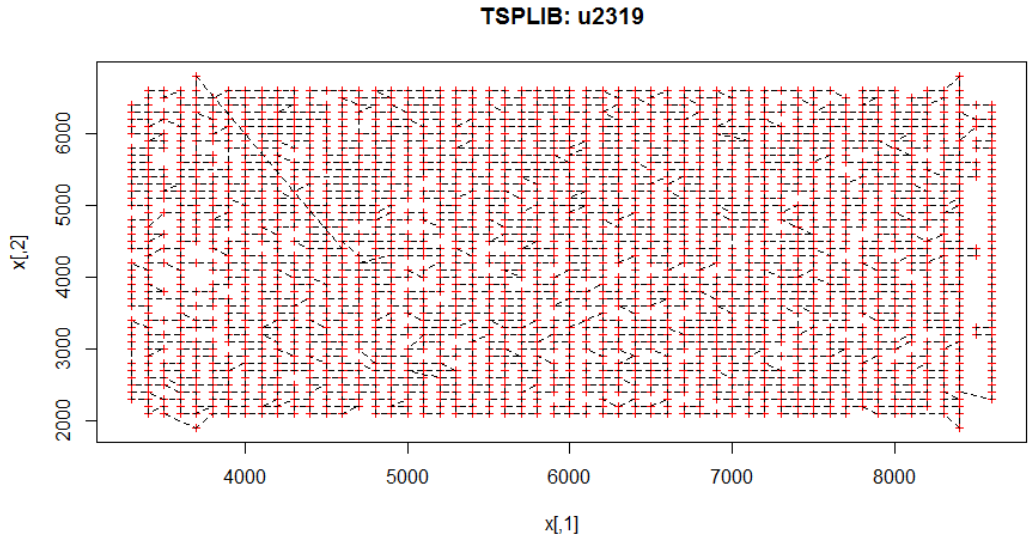
, gdzie $x_1 \in [-5, 10]$ oraz $x_2 \in [0, 15]$.



Rysunek 1: Jakość wyników

Z wykresu (rys. 1) wynika, że funkcja ta ma stosunkowo duży obszar w którym może znajdować się minimum oraz dwie strefy w których wartości są dużo większe.

Na kolejnych stronach zamieszczono wyniki pomiarów dla różnych wartości parametrów algorytmu genetycznego. Kolejno dokonano pomiarów dla różnych wartości: prawdopodobieństwa mutacji i krzyżowania, wielkości populacji, ilości iteracji oraz elityzmu. Wszystkie pomiary wykonano dla 4 różnych ustawień domyślnych parametrów (serie 1 – 4).



Rysunek 2: Wartość znalezionego minimum funkcji Branin w zależności od prawdopodobieństwa mutacji

Rysunek 3: Wartość znalezionego minimum funkcji Branin w zależności od prawdopodobieństwa krzyżowania

Na wykresie (rys. 2) można zauważyć niski wpływ ustawienia mutacji na znalezione rozwiązania. Przy wszystkich parametrach domyślnych funkcja znajduje się w pobliżu optymalnej wartości. Miejscowe odchylenia są tu najprawdopodobniej związane z charakterem algorytmu i zbyt małą ilością prób poddanych uśrednieniu. Nie możemy tutaj określić czy przy wyłączonej zarówno mutacji jak i krzyżowaniu wyniki ulegają pogorszeniu, gdyż nie ma w tym obszarze spójności. Podobne wnioski możemy wskazać dla wykresu krzyżowania (rys. 3).

Rysunek 4: Wartość znalezionego minimum funkcji Branin w zależności od rozmiaru populacji

Rysunek 5: Wartość znalezionego minimum funkcji Branin w zależności od ilości iteracji

Z wykresu (rys. 4) można odczytać podatność funkcji na zmiany rozmiaru populacji. Wyniki zbliżone do oczekiwanych zostały uzyskane dla wartości wynoszącej 45 jednostek. Widać również, że przy małej populacji znaczenie mutacji i krzyżowania jest większe. Zauważalny jest wzrost jakości rozwiązania wraz ze wzrostem ilości jednostek populacji.

Wykres (rys. 5) wskazuje wyraźną zmianę jakości rozwiązań dla 60 i więcej iteracji. Poniżej tej wartości uzyskiwane wyniki są niestabilne, powyżej osiągają wartość zbliżoną do oczekiwanej szczególnie dla serii 4 (czyli z włączoną mutacją i krzyżowaniem).

Rysunek 6: Wartość znalezionego minimum funkcji Branin w zależności od przyjętego elityzmu

Z wykonanych pomiarów (rys. 6) wynika, że dla uzyskania optymalnego rozwiązania należy zastosować wartość elityzmu na poziomie przynajmniej 0,35. Jego ustawienie poniżej tej wartości powoduje obniżenie się jakości rezultatów.

Rysunek 7: Poglądowa lokalizacja najlepszego znalezionego minimum funkcji Branin dla pomiarów przy zmianach elityzmu

4 Podsumowanie

W trakcie prowadzonych badań przetestowano algorytm genetyczny w zadaniu optymalizacji dla 9 funkcji testowych. Analizie poddano wpływ zmiany każdego z parametrów dla 4 różnych konfiguracji pozostałych wartości domyślnych.

Wartość prawdopodobieństwa mutacji i krzyżowania zdaje się odgrywać drugorzędną rolę. Istotne jednak by chociaż jedna z nich była włączona z prawdopodobieństwem większym niż 0.

Najlepszym ustawieniem dla elityzmu jest prawdopodobieństwo rzędu 0,5.

Z pewnością należałoby zwiększyć ilość prób poddawanych uśrednianiu gdyż dla przyjętych 20 wyniki ciągle są niestabilne. Warto by również rozważyć pomijanie kilku najlepszych i najgorszych wyników przed uśrednianiem.

Co ciekawe wyniki są widocznie gorsze przy konfiguracji w której krzyżowanie jest wyłączone a p. mutacji wynosi 0,5. Taka prawidłowość objawia się dla wszystkich badanych funkcji.

Literatura

[1] Artur Suchwałko „Wprowadzenie do R dla programistów innych języków” https://cran.r-project.org/doc/contrib/R-dla-programistow-innych-jezykow.pdf

[2] Luca Scrucca „Package GA” https://cran.r-project.org/web/packages/GA/GA.pdf

[3] Surjanovic, S. & Bingham, D. (2013). „Virtual Library of Simulation Experiments: Test Functions and Datasets.” Retrieved April 3, 2017, from http://www.sfu.ca/ ssurjano.

[4] Momin Jamil, Xin-She Yang „A literature survey of benchmark functions for global optimization problems”, Int. Journal of Mathematical Modelling and Numerical Optimisation, Vol. 4, No. 2, pp. 150–194. (2013)

[5] Ajith Abraham, Aboul-Ella Hassanien, Patrick Siarry, Andries Engelbrecht, „Foundations of Computational Intelligence Volume 3” (2009)

[6] Onay Urfalioglu, Orhan Arikan „Self-adaptive randomized and rank-based differential evolution for multimodal problems” (2011)