WSI – ćwiczenie 2.

Algorytmy ewolucyjne i genetyczne

grupa 101

Magdalena Kotynia

23 marca 2021

1. Cel ćwiczenia

W ramach drugiego cwiczenia należało zaimplementować metodę realizujacą algorytm genetyczny w wersji Hollanda dla wektorów binarnych. Następnie należało zbadać działanie zaimplementowanego algorytmu na problemie znalezienia wektora 6 liczb całkowitych z zakresu [−4, 3] maksymalizujących funkcję *f:*

$$f(x) = -\frac{\sum_{i=1}^{6}\left(x_i^4 - 16x_i^2 + 5x_i\right)}{2}$$

Należało również zbadać wpływ rozmiaru populacji na działanie algorytmu.

2. Implementacja algorytmu

Algorytm zaimplementowano w języku R w sposób przdstawiony na Rysunku 2., według pseudokodu algorytmu gentycznego, przedstawionego na Rysunku 1. W implementacji zastosowano identyczne oznaczenia parametrów z oznaczeniami w Pseudokodzie. Dodatkowe parametry zastosowane na potrzebę rozwiązania problemu postawionego w Ćwiczeniu 2. to:

limit – dopuszczalna liczba elementów w wektorze liczb całkowitych maksymalizujących funkcję *f.* (W przypadku tego zadania *limit = 6.*)

range – zakres dopuszczalnych liczb w wektorze liczb całkowitych maksymalizujących funkcję *f.* (W przypadku tego zadania *range = [-4,3].*)

fun – funkcja celu (zamiast q(x) z pseudokodu)

Jako sposób reprodukcji zaimplementowano reprodukcję turniejową z dwuosobową grupą turniejową. Ocena osobnika polegała na obliczeniu dla niego wartości maksymalizowanej funkcji. Jeśli liczba jedynek w osobniku, świadczących o występowaniu liczby na danej pozycji, była różna od sześciu, osobnik był oceniany na zero.

Pseudokod algorytmu genetycznego

Data: $q(x), P_0, \mu, p_m, p_c, t_{max}$
Result: $\hat{x}^*$

begin
  $t \leftarrow 0$
  $o \leftarrow$ ocena(q, P_0)
  $\hat{x}^* \leftarrow$ znajdź najlepszego(P_0, o)
  while $t < t_{max}$ **do**
    $R \leftarrow$ reprodukcja(P_t, o, μ)
    $M \leftarrow$ krzyżowanie i mutacja(R, p_m, p_c)
    $o \leftarrow$ ocena(q, M)
    $x_t^* \leftarrow$ znajdź najlepszego(M, o)
    if $x_t^* < \hat{x}^*$ **then**
      $\hat{x}^* \leftarrow x_t^*$
    end
    $P_{t+1} \leftarrow M$
    $t \leftarrow t + 1$
  end
end

Parametry algorytmu genetycznego

- $q(x)$ – funkcja celu
- P_0 – populacja początkowa
- μ – liczba osobników w populacji
- p_m – prawdopodobieństwo mutacji
- p_c – prawdopodobieństwo krzyżowania. Dla każdej pary, jak $U(0,1) < p_c$ to stosujemy krzyżowanie, np. w schemacie 2 osobniki potomne z 2 rodziców. W przeciwnym razie do następnego kroku przechodzą 2 osobniki wybrane na rodziców.
- t_{max} – maksymalna liczba iteracji (pokoleń)

Rysunek 1. Pseudokod algorytmu genetycznego. Źródło: Rafał Biedrzycki, Wprowadzenie do Sztucznej Inteligencji (WSI). Notatki wygenerowane automatycznie ze slajdów. ©2021

```r
geneticAlgorithm <- function(fun, mi, pm, pc, tmax, limit, range) {
  P0 <- createPopulation(mi, range)
  t <- 0
  evaluation <- evaluate(fun, P0, range, limit)
  meanEvaluation <- matrix()
  best <- findBest(P0, evaluation)
  Pt <- P0
  while (t < tmax){
    R <- tournament(Pt, evaluation, mi, size = 6)
    C <- crossingOver(R, pc)
    M <- mutation(C, pm)
    evaluation <- evaluate(fun, M, range, limit)
    meanEvaluation[t+1] <- mean(unlist(evaluation))
    new_best <- findBest(M, evaluation)
    if (computeFunVal(f, best, range) < computeFunVal(f,new_best, range)){
      best <- new_best
    }
    Pt <- M
    t <- t + 1
  }
  return(list(best, meanEvaluation))
}
```

Rysunek 2. Kod algorytmu genetycznego w języku R.

3. Badanie algorytmu

W Ćwiczeniu sprawdzono wpływ rozmiaru populacji na działanie algorytmu przy stałych pozostałych parametrach wejściowych. Przyjęte wartości parametrów wejściowych:

pm - 0.1

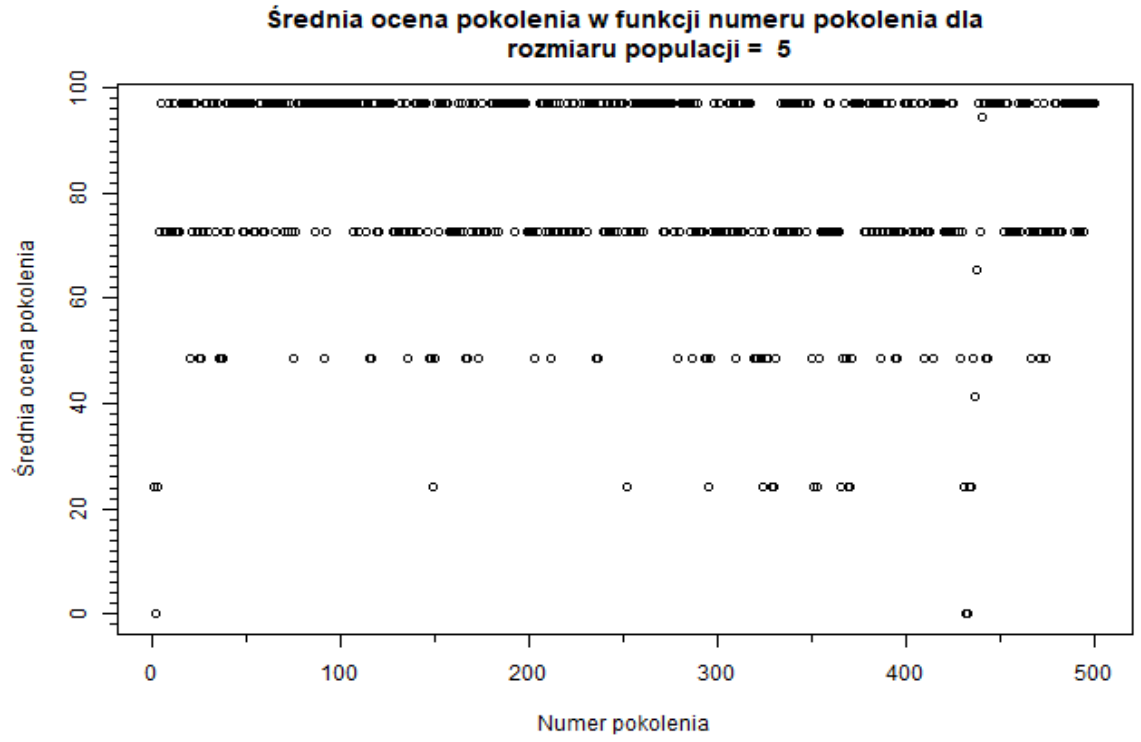
pc – 0.7

tmax – 300

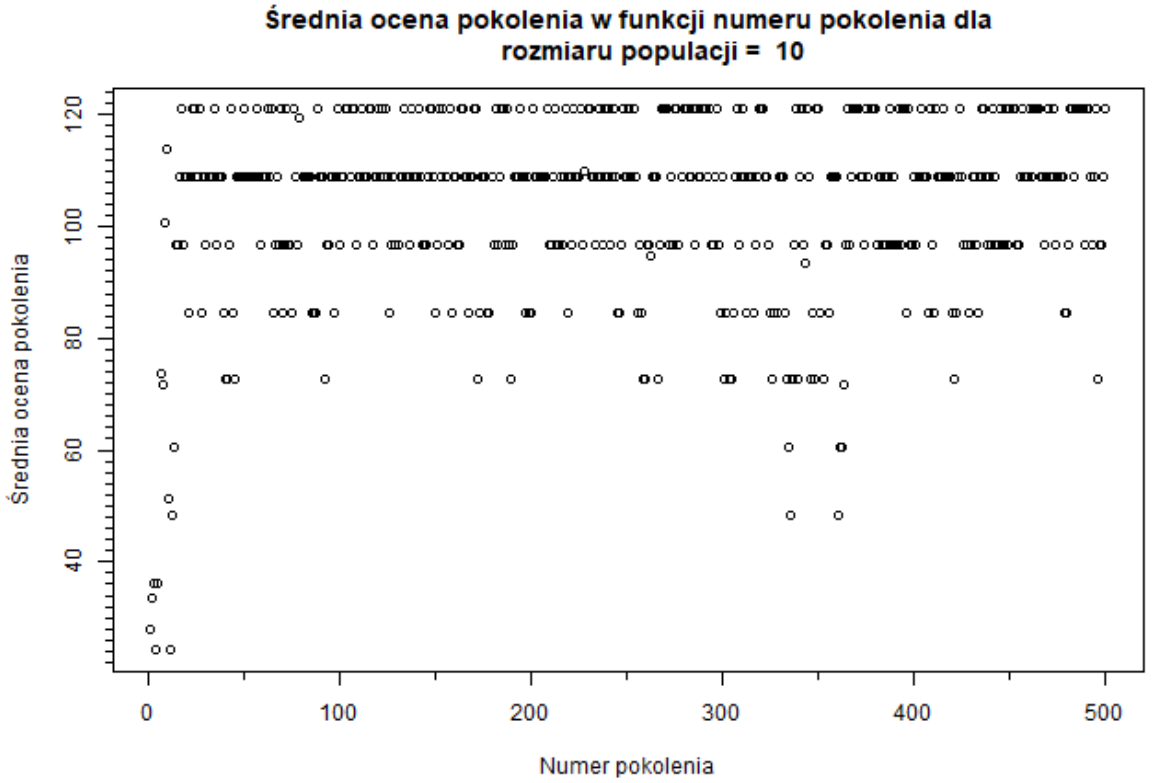
Szukany wektor maksymalizujący rozwiązanie to wektor [1 1 1 1 0 0 1 1], gdzie 1 oznacza, wystąpienie liczby na tej samej pozycji z wektora [-4 -3 -2 -1 0 1 2 3], czyli wynikiem jest wektor x = [-4, -3, -2, -1, 2, 3], który dają wynik funkcji *f(x)* = 131. Tabela 1. przedstawia, jaka maksymalna wartość, i w którym numerze pokolenia została osiągnięta dla różnych rozmiarów populacji. Dla każdego rozmiaru, z wyjątkiem 15 i 20, zostało osiągnięte maksimum. Dla rozmiarów 30, 40, 50, 70, 100 wektor będący rozwiązaniem problemu musiał znajdować się już w populacji początkowej. Można zauważyć pewną losowość w szysbkości znalezienia rozwiązania, prawdopodobnie uwarunkowaną specyfiką rekombinacji, mutacji i krzyżowania, które zachodzą z pewnym prawdopodobieństwem. Rysunki 3. – 11. przedstawiają zależność średniej oceny pokolenia w funkcji numeru pokolenia, każdy Rysunek dla innego rozmiaru populacji. Można zauważyć, że im większa populacja, tym większą średnią wartość oceny osiągają pokolenia. Widać również, że wraz ze wzrostem rozmiaru populacji maleje rozrzut średnich ocen pokoleń. Można po tym wnioskować, że w większej populacji słabsze osobniki są szybciej eliminowane przez te silniejsze.

Tabela 1. Numer pokolenia, dla którego osiągnięto pierwszy raz maksymalną wartość w funkcji rozmiaru populacji.

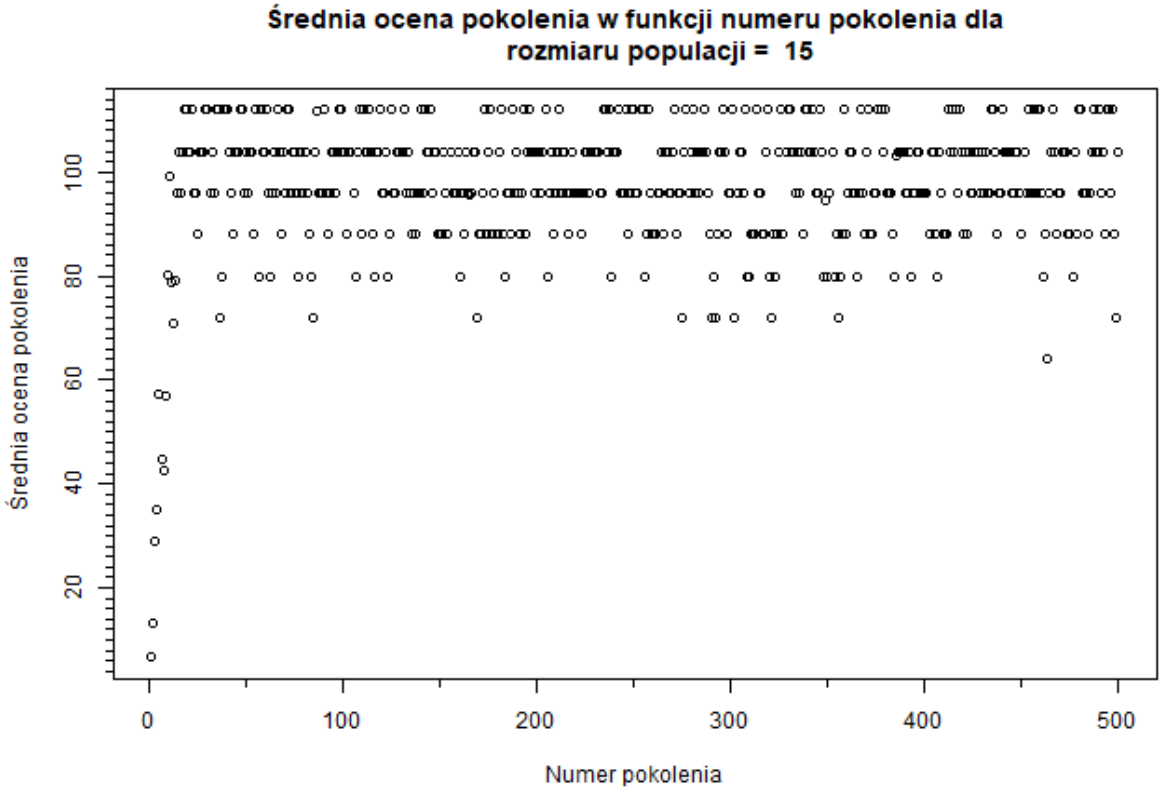
Rozmiar populacji	Max (Numer pokolenia)
5	131 (371)
10	131 (7)
15	126 (4)
20	126 (443)
30	131 (1)
40	131 (1)
50	131 (1)
70	131 (1)
100	131 (1)



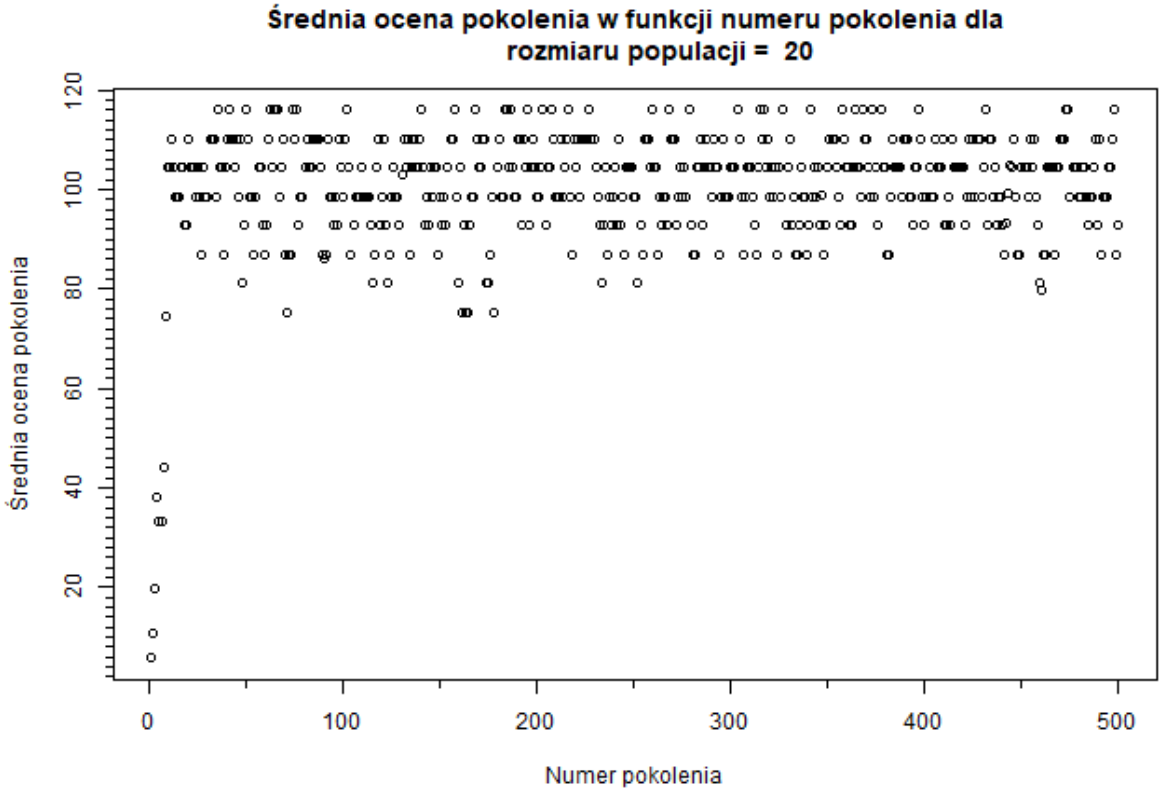
Rysunek 3.



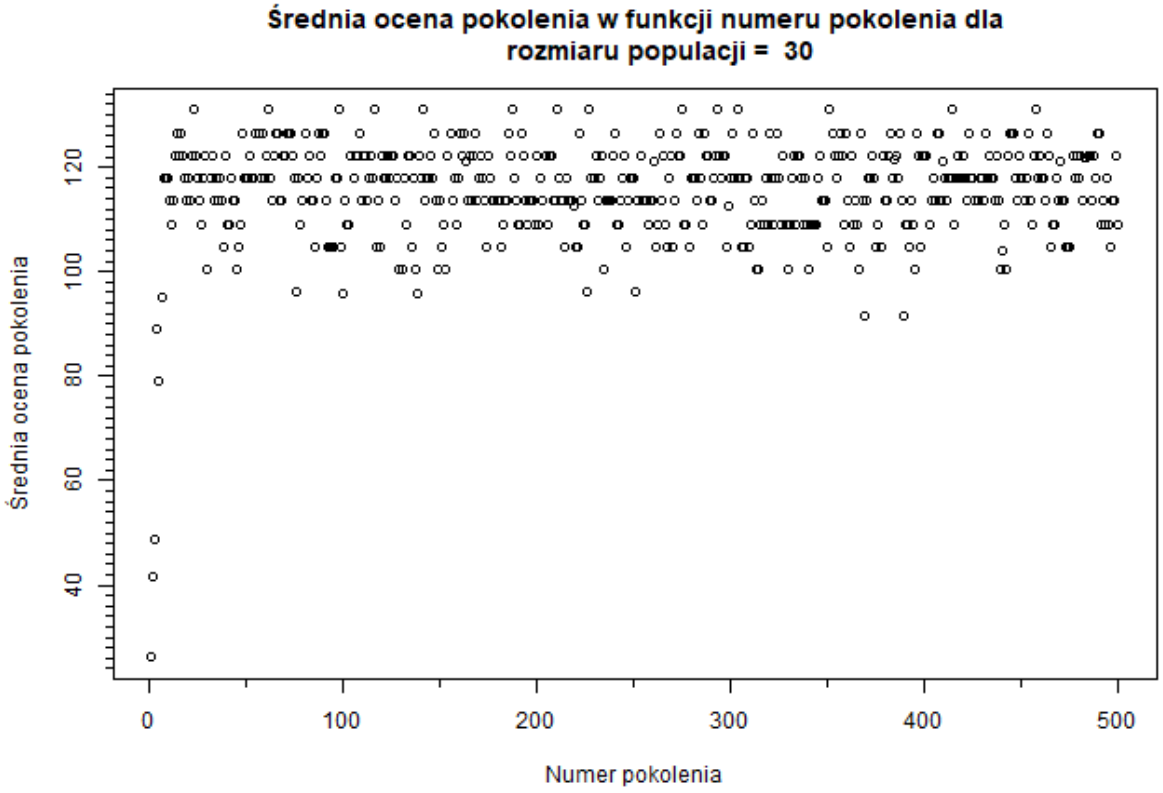
Rysunek 4.



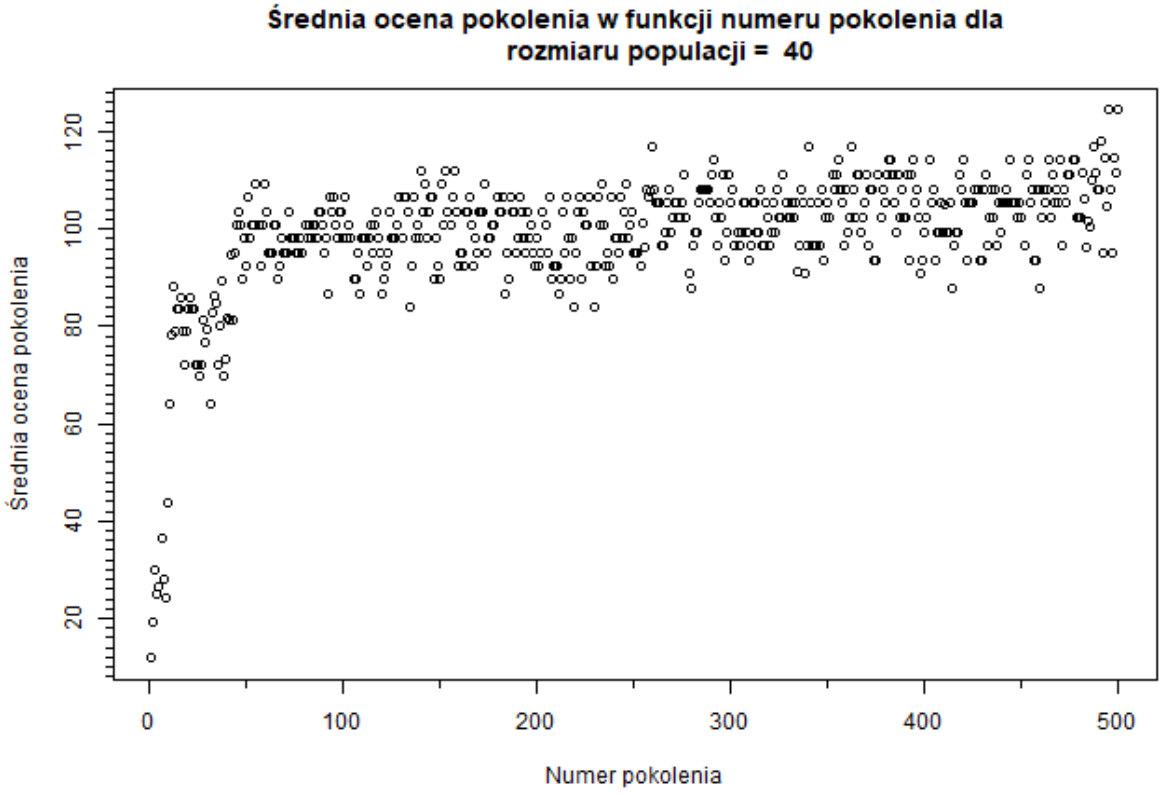
Rysunek 5.



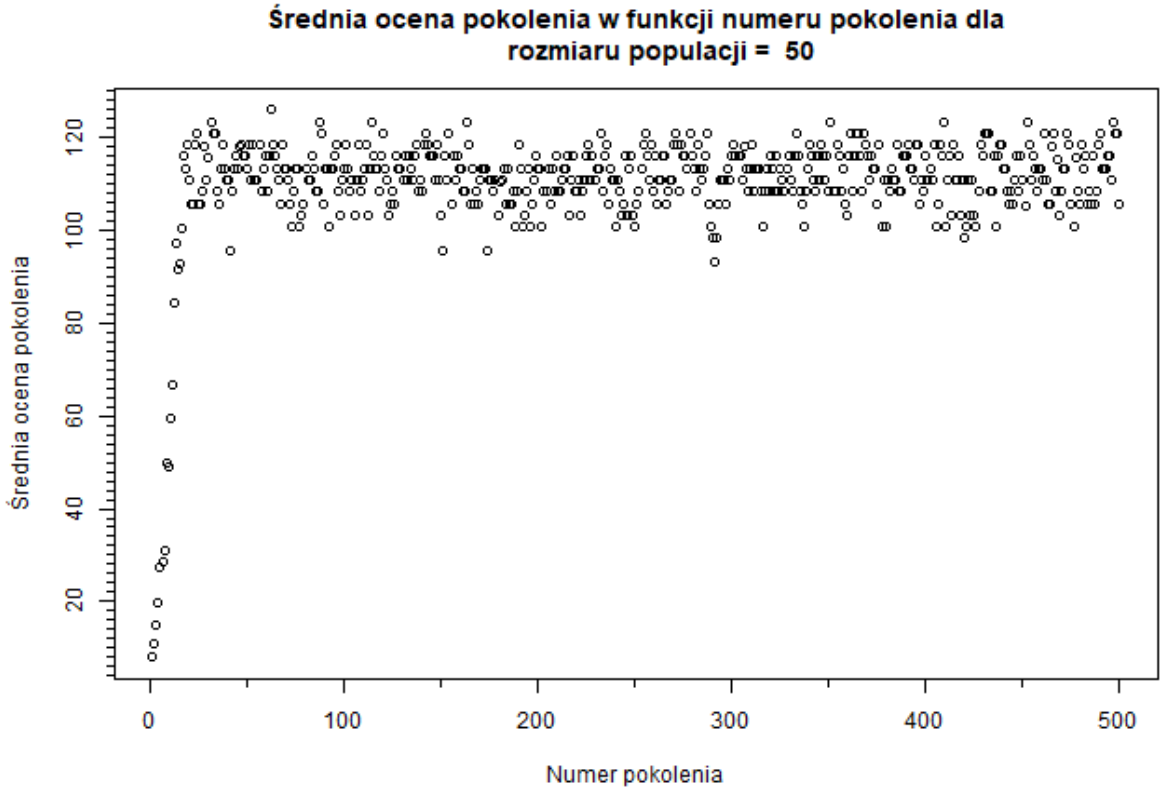
Rysunek 6.



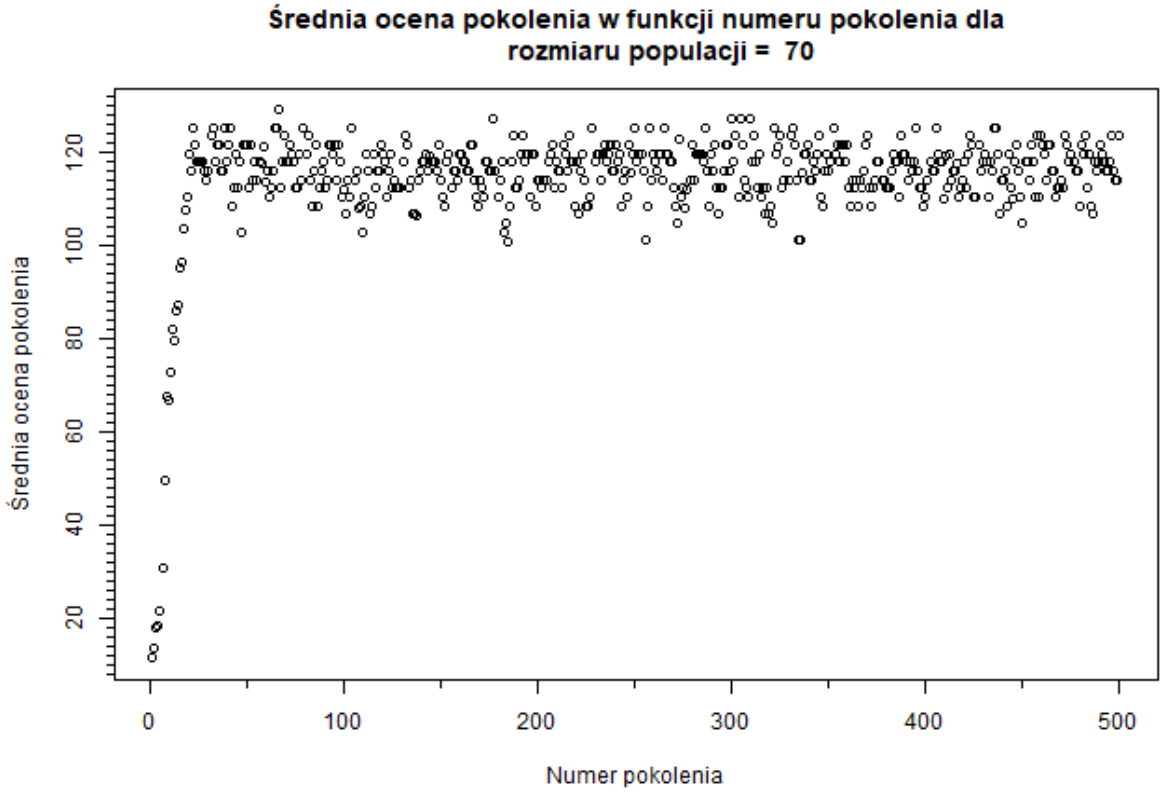
Rysunek 7.



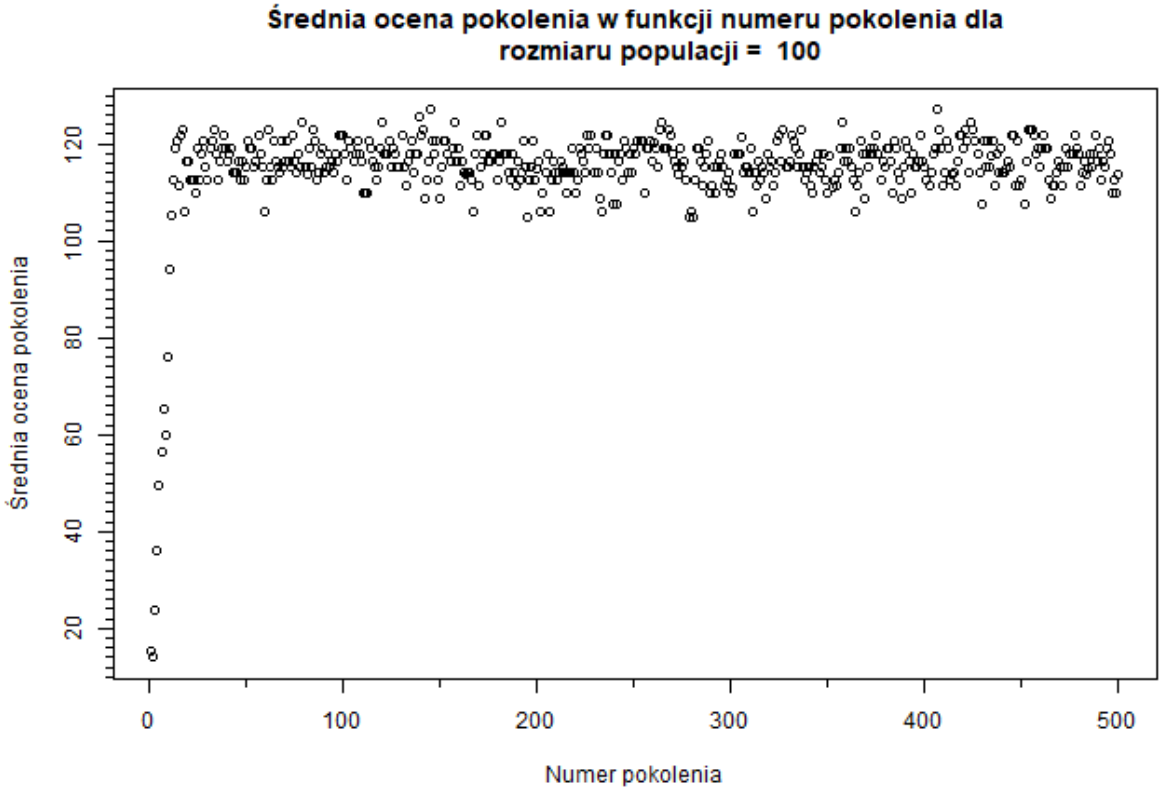
Rysunek 8.



Rysunek 9.



Rysunek 10.



Rysunek 11.

4. Wnioski

Dla danego problemu algorytm jest skuteczny również przy małym rozmiarze populacji, jednak może się zdarzyć „utknięcie" algorytmu w punkcie bliskim rozwiązaniu, jak to widać w tym przypadku dla rozmiarów populacji 15 i 20. W tych przypadkach być może algorytm znalazłby rozwiązanie dla większej liczby pokoleń niż 500, lub przy zwiększeniu prawdopodobieństwa mutacji. Są to istotne parametry, indywidualnie dobierane dla różnych problemów, często eksperymentalnie.

5. Kod algorytmu

```
library(mosaic)
library(R.utils)
library(Hmisc)

f <- makeFun(-sum(x ^ 4 - 16 * x ^ 2 + 5 * x) / 2 ~ x)


geneticAlgorithm <- function(fun, mi, pm, pc, tmax, limit, range) {
  P0 <- createPopulation(mi, range)
  t <- 0
  evaluation <- evaluate(fun, P0, range, limit)
  meanEvaluation <- matrix()
  bestInIteration <- matrix()
  best <- findBest(P0, evaluation)
  Pt <- P0
  while (t < tmax){
```

```
    meanEvaluation[t+1] <- mean(unlist(evaluation))
    bestInIteration[t+1] <- computeFunVal(f, best, range)
    R <- tournament(Pt, evaluation, mi, 2)
    C <- crossingOver(R, pc)
    M <- mutation(C, pm)
    evaluation <- evaluate(fun, M, range, limit)
    new_best <- findBest(M, evaluation)
    if (computeFunVal(f, best, range) < computeFunVal(f,new_best, range)){
      best <- new_best
    }
    Pt <- M
    t <- t + 1
  }
  bestVector <- best*range
  return(list(bestVector, meanEvaluation, bestInIteration))
}


computeFunVal <- function(fun, binary, range){
  which_numbers <- binary * range
  funVal <- fun(which_numbers)
  return(funVal)
}


evaluate <- function(fun, population, range, limit) {
  which_numbers <- lapply(population, function(x)
    x * range)
  y <- lapply(which_numbers, fun)
  num_elements <- lapply(population, sum)
  y[num_elements > limit] = 0
  y[num_elements < limit] = 0
  return(y)
}


findBest <- function(population, evaluation) {
  best_idx <- match(max(as.numeric(evaluation)), evaluation)
  best <- population[[best_idx]]
}


createPopulation <- function(mi, range) {
  set.seed(10)
  emptyPopulation <- vector(mode = "list", length = mi)
  population <- lapply(emptyPopulation,
                       function(x)
                         x <- round(runif(n = length(range))))
  set.seed(Sys.time())
  return(population)
}


tournament <- function(population, evaluation, mi, tournament_size) {
  new_population <- list()
  for (i in 1:mi) {
    rivals_idxs <- match(sample(population, tournament_size,
                                replace = TRUE), population)
    rivals_evaluations <-
      lapply(rivals_idxs, function(x)
        evaluation[[x]])
```

```
    winner <-  max(as.numeric(rivals_evaluations))
    winner_idx <- rivals_idxs[[match(winner, rivals_evaluations)]]
    new_population[[i]] <- population[[winner_idx]]
  }
  return(new_population)
}

crossingOver <- function(population, pc){
  new_population <- list()
  population <- sample(population, length(population), replace = FALSE)
  for (i in 1:(length(population)/2)){
    chromosome1 <- population[[i]]
    chromosome2 <- population[[i+length(population)/2]]
    if (runif(1) < pc){
      crossPlace <- sample(c(1:(length(chromosome1)-1)), 1)
      new_chromosome1 <- c(chromosome1[1:crossPlace],
                           chromosome2[(crossPlace+1):length(chromosome2)])
      new_chromosome2 <- c(chromosome2[1:crossPlace],
                           chromosome1[(crossPlace+1):length(chromosome1)])
      new_population[[i]] <- new_chromosome1
      new_population[[i+length(population)/2]] <- new_chromosome2
    }
    else{
      new_population[[i]] <- chromosome1
      new_population[[i+length(population)/2]] <- chromosome2
    }
  }
  return(new_population)
}

mutation <- function(population, pm){
  new_population <- list()
  population <- sample(population, length(population), replace = FALSE)
  isMutated <- runif(length(population))<pm
  toMutate <- population[isMutated]
  new_population <- population[!isMutated]
  geneIdxs <- sample(c(1:length(population[[1]])), length(toMutate),
replace = TRUE)
    if (length(toMutate)>0){
    for (i in 1:length(toMutate)){
      toMutate[[i]][[geneIdxs[i]]] <- !toMutate[[i]][[geneIdxs[i]]]
    }
  }
  new_population <- c(new_population, toMutate)
  return(new_population)
}

# testy
range <- c(-4:3)
populationSizes <- c(5,10,15,20,30,40,50,70,100)
bestVectors3 <- list()
meanEvaluations3 <- list()
bestsInIteration3 <- list()
for (i in 1:length(populationSizes)){
  result <- geneticAlgorithm(f, populationSizes[i], 0.1, 0.7, 500, 6,
range)
  bestVectors3[[i]] <- result[[1]]
  meanEvaluations3[[i]] <- result[[2]]
  bestsInIteration3[[i]] <- result[[3]]
}
```

```r
# ploty
for (i in 1:length(populationSizes)){
  png(file=paste("pop_size_",populationSizes[i],".png", collapse = NULL),
      width = 650, height = 450)
  plot(1:500, meanEvaluations3[[i]],
       main = paste("Średnia ocena pokolenia w funkcji numeru pokolenia dla
       rozmiaru populacji = ",populationSizes[i]),
       xlab = "Numer pokolenia",
       ylab = "Średnia ocena pokolenia")
  minor.tick(ny=10)
  dev.off()
}
```