Algorytm ewolucyjny z selekcją turniejową i sukcesją elitarną

Autor: Jan Retkowski

Importowanie bibliotek

```
library(ggplot2)
library(reshape2)
library(gridExtra)
library(ggthemes)
library(microbenchmark)

set.seed(0)
```

Implementacja algorytmu i funkcji pomocniczych

Pierwszym krokiem jest zdefiniowanie funkcji `optimize.evolution`, która realizuje **algorytm ewolucyjny** z **selekcją turniejową** i **sukcesją elitarną**.

```
optimize.evolution <- function(fitness_function, pop_size, starting_distribution,
                               mutation_chance, mutation_strength, elite_size,
                               tournament_size, budget, maximize = T, save_path = F) {



    if (!maximize)
        fitness_func_internal <- function(a) -fitness_function(a)
    else
        fitness_func_internal <- fitness_function

    population <- evolution.population(pop_size, starting_distribution)
    population$fitness <- mapply(fitness_func_internal, population$specimen)

    generations <- budget %/% pop_size

    if (save_path) {
        path_history <- list(population$specimen[[which.max(population$fitness)]])
    }

    for (i in 1:generations) {
        parent.pop <- evolution.tournament_select(population, pop_size, tournament_size, elite_size)
        offspring.pop <- evolution.mutate(parent.pop, mutation_chance, mutation_strength)
        elite.pop <- evolution.elitist(population, elite_size)

        population$specimen <- c(elite.pop, offspring.pop)

        population$fitness <- mapply(fitness_func_internal, population$specimen)
```

```
        if (save_path)
            path_history <- c(path_history, list(population$specimen[[which.max(population$fitness)]]))

    }

    best_specimen <- population$specimen[[which.max(population$fitness)]]

    result <- list(best_specimen = best_specimen, best_fit = fitness_function(best_specimen),
                   mean_fit = if (maximize) mean(population$fitness) else -mean(population$fitness),
                   path_history =  if (save_path) path_history else NULL)

    return(result)
}


evolution.population <- function(pop_size, starting_distribution) {
    population <- list()
    population$specimen <- lapply(vector("list", length = pop_size), starting_distribution)

    return(population)
}


evolution.tournament_select <- function(population, pop_size, tournament_size, elite_size) {
    parent.pop <- vector("list", length = pop_size - elite_size)

    for (i in 1:(pop_size - elite_size)) {
        indexes <- sample(length(population$specimen), tournament_size, replace = T)

        parent.pop[[i]] <- population$specimen[[which.max(population$fitness[indexes])]]
    }

    return(parent.pop)
}


evolution.mutate <- function(parent.pop, mutation_chance, mutation_strength) {
    offspring.pop <- vector("list", length = length(parent.pop))

    for (i in 1:length(parent.pop)) {

        if (sample(c(T, F), 1, prob = c(mutation_chance, 1 - mutation_chance))) {
            offspring.pop[[i]] <-  (parent.pop[[i]] + matrix(rnorm(length(parent.pop[[i]]), 0,
                                                                   mutation_strength), ncol = 1))

        } else {
            offspring.pop[[i]] <- parent.pop[[i]]
        }
    }
    return(offspring.pop)
}
```

```
evolution.elitist <- function(population, elite_size) {
    elite.pop <- population$specimen[sort(population$fitness, decreasing = T,
                                          index.return = T)$ix[1:elite_size]]

    return(elite.pop)
}
```

Drugim krokiem jest zdefiniowanie funkcji pomocniczej `prepare.plot2D`, która zostanie wykorzystana do wizualizacji działania algorytmu.

```
prepare.plot2D <- function(func, x_from, x_to, y_from, y_to, breaks = 100) {

    graph <- as.data.frame(expand.grid(seq(x_from, x_to, length.out = breaks),
                                       seq(y_from, y_to, length.out = breaks)))

    graph$z <- mapply(function(a, b) func(matrix(c(a, b), ncol = 1)), graph$Var1, graph$Var2)

    g <- ggplot(data = graph, aes(x = Var1, y = Var2, fill = z, z = z)) +
        geom_raster() +
        scale_fill_viridis_c() +
        geom_contour(color = "black", size = 0.2) +
        # geom_density2d() +
        theme_wsj() +
        labs(x = "x", y = "y", fill = "z") +
        theme(legend.position = "right", legend.direction = "vertical", axis.title = element_text())

    return(g)
}
```

Druga funkcja pomocnicza to `test.algorithm`, która posłuży do testowania średniej wartości dopasowania populacji dla n różnych seed'ów.

```
test.algorithm <- function(algorithm, times) {
    for (seed in 0:(times - 1)) {
        set.seed(seed)
        cat("seed = ", seed, ", mean fitness = ", algorithm()$mean_fit, "\n")
    }
}
```

Ostatnia funkcja pomocnicza to `examine.elite_size`, która posłuży badaniu wpływu rozmiaru elity `elite_size` na działanie algorytmu.

```
examine.elite_size <- function(algorithm, elite_size_range, seed = 0) {

    for (elite_size in elite_size_range) {
        set.seed(seed)
        cat("elite_size = ", elite_size,
            ", mean fitness = ",algorithm(elite_size)$mean_fit, "\n")
    }
}
```

Testowanie algorytmu

Maksymalizacja funkcji nr 1 Nastpępnie można przystąpić do testowania algorytmu. Pierwszą funkcją testową jest $f_1(x) = \varphi(x, \mu_1, \Sigma_1) + \varphi(x, \mu_2, \Sigma_2) + \varphi(x, \mu_3, \Sigma_3)$, gdzie $\varphi(x, \mu, \Sigma) = \frac{e^{(-0.5(x-\mu)^\intercal)\Sigma^{-1}(x-\mu)}}{\sqrt{(2\pi)^{dim(x)*|\Sigma)|}}}$ jest funkcją gęstości prawdopodobieństwa rozkładu normalnego o wektorze średnich μ i macierzy kowariancji Σ. Funkcja ta ma zostać zmaksymalizowana, a średnia wartość dopasowania populacji `mean_fit` powinna dla większości seed'ów przekraczać próg 0.15. Maksymalny budżet wynosi 1e+6 wywołań funkcji celu. W tym wypadku pożądany rezultat jest możliwy do osiągnięcia przy budżecie 1e+5, czyli o rząd wielkości niższym od maksymalnego. Dzieki temu czas wykonania algorytmu jest znacznie krótszy. Ponieważ testom ma podlegać klasyczna wersja algorytmu ewolucyjnego, szansa mutacji `mutation_chance` zostanie ustawiona na `1`.

```
normal2D <- function(x, mu, sigma)  {

    exp(-0.5 %*% t(x - mu) %*% solve(sigma) %*% (x - mu)) / sqrt((2 * pi)^length(x) * det(sigma))
}

func <- function(x, mu_1, sigma_1, mu_2, sigma_2, mu_3, sigma_3) {

    normal2D(x, mu_1, sigma_1) + normal2D(x, mu_2, sigma_2) + normal2D(x, mu_3, sigma_3)
}

mu_1 <- matrix(c(14, -11), ncol = 1)
sigma_1 <- matrix(c(1.3, -0.5, -0.5, 0.8), ncol = 2)

mu_2 <- matrix(c(10, -10), ncol = 1)
sigma_2 <- matrix(c(1.7, 0.4, 0.4, 1.2), ncol = 2)

mu_3 <- matrix(c(7, -13), ncol = 1)
sigma_3 <- matrix(c(1, 0, 0, 1.5), ncol = 2)



fitness_func <- function(x) func(x, mu_1, sigma_1, mu_2, sigma_2, mu_3, sigma_3)
starting_dist <- function(x = NULL) matrix(c(rnorm(1, 0 , 1), rnorm(1, 0, 1)), ncol = 1)

test.algorithm(function() optimize.evolution(
    fitness_func = fitness_func,
    pop_size = 10,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 2,
    elite_size = 9,
    tournament_size = 7,
    budget = 1e+5), times = 10)
```

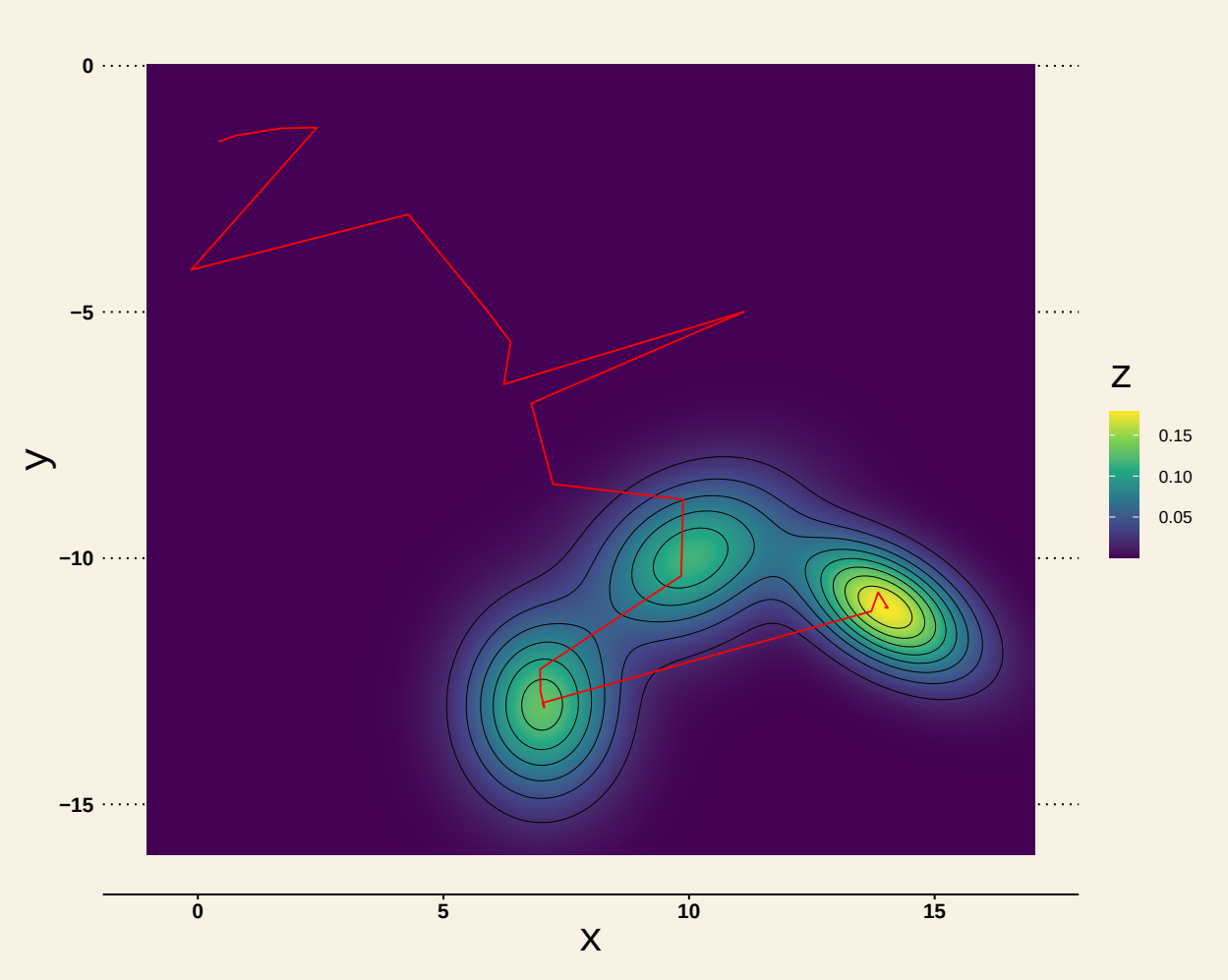
```
## seed =  0 , mean fitness =  0.1645727
## seed =  1 , mean fitness =  0.1647928
## seed =  2 , mean fitness =  0.1616753
## seed =  3 , mean fitness =  0.1609053
## seed =  4 , mean fitness =  0.1615685
## seed =  5 , mean fitness =  0.161295
## seed =  6 , mean fitness =  0.1620946
## seed =  7 , mean fitness =  0.160943
## seed =  8 , mean fitness =  0.1620287
## seed =  9 , mean fitness =  0.1610073
```

Jak widać algorytm przekroczył pożądany próg dla wszystkich 10 seed'ów. Na tej podstawie można stwierdzić, że funkcja działa poprawnie.

Teraz działanie algorytmu zostanie zwizualizowane za pomocą wykresu.

```
set.seed(0)
path_history <- optimize.evolution(
    fitness_func = fitness_func,
    pop_size = 10,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 2,
    elite_size = 9,
    tournament_size = 7,
    budget = 1e+5,
    save_path = T)$path_history

prepare.plot2D(fitness_func, -1, 17, -16, 0, 300) +
    annotate("function", x = mapply(function(a) a[[1]], path_history),
             y = mapply(function(a) a[[2]], path_history), color = "red", size = 0.5)
```



Wykres pokazuje najlepszego osobnika z każdej generacji. Po drodze odkrywa on dwa optima lokalne, jednak dzięki dużej sile mutacji udaje mu się z nich łatwo wydostać i szybko zbliżyć się do optimum globalnego.

Po wizualizacji zbadany zostanie wpływ parametrów na działanie algorytmu ewolucujnego. W tym przypadku testom podlega parametr k **sukcesji elitarnej**, w tym wypadku nazwany `elite_size`.

```
examine.elite_size(function(k) optimize.evolution(
    fitness_func = fitness_func,
    pop_size = 10,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 2,
    elite_size = k,
    tournament_size = 7,
    budget = 1e+5),
    elite_size_range = 1:9)
```

```
## elite_size =  1 , mean fitness =  0.04223606
## elite_size =  2 , mean fitness =  0.06312089
## elite_size =  3 , mean fitness =  0.06438228
## elite_size =  4 , mean fitness =  0.09340317
## elite_size =  5 , mean fitness =  0.09763776
## elite_size =  6 , mean fitness =  0.1147937
## elite_size =  7 , mean fitness =  0.1320996
## elite_size =  8 , mean fitness =  0.1484523
## elite_size =  9 , mean fitness =  0.1645727
```

Dla funkcji $f_1(x)$ zwiększanie parametru k sukcecji elitarnej powoduje polepszenie się średniego wyniku. Dzięki zwiększeniu rozmiaru elity więcej najlepszych osobników przechodzi do następnej generacji. W ten sposób kosztem eksploracji zwiększana jest eksploatacja przestrzeni przeszukiwań. Jako, że ta funkcja nie posiada wielu ekstremów lokalnych, zmniejszenie eksploracji nie powoduje problemów. Natomiast dzięki większej eksploatacji algorytm znacznie szybciej zbliża się do optimum globalnego, gdy już znajduje się w jego okolicach.

Minimalizacja funkcji nr 2 Drugą funkcją testową jest $f_2(x) = -20e^{-0.2\sqrt{x^\intercal x}} - e^{0.5\cos(2\pi x_1)+cos(2\pi x_2)} + e + 20$. Funkcja ta ma zostać zminimalizowana, a średnia wartość dopasowania populacji `mean_fit` powinna dla większości seed'ów być poniżej progu 1. Maksymalny budżet wynosi 1e+5 wywołań funkcji celu. Również w tym przypadku szansa mutacji `mutation_chance` zostanie ustawiona na `1`.

```
fitness_func <- function(x) {

    (-20 * exp(-0.2 * sqrt(0.5 * t(x) %*% x))
     - exp(0.5 * (cos(2 * pi * x[1]) + cos(2 * pi * x[2])))
     + exp(1) + 20)
}

starting_dist <- function(x = NULL) matrix(c(rnorm(1, 3 , 1), rnorm(1, 3, 1)), ncol = 1)


test.algorithm(function () optimize.evolution(
    fitness_function = fitness_func,
    pop_size = 2000,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 0.22,
    elite_size = 1500,
```

```
    tournament_size = 3,
    budget = 1e+5,
    maximize = F,
    save_path = F), times = 10)
```

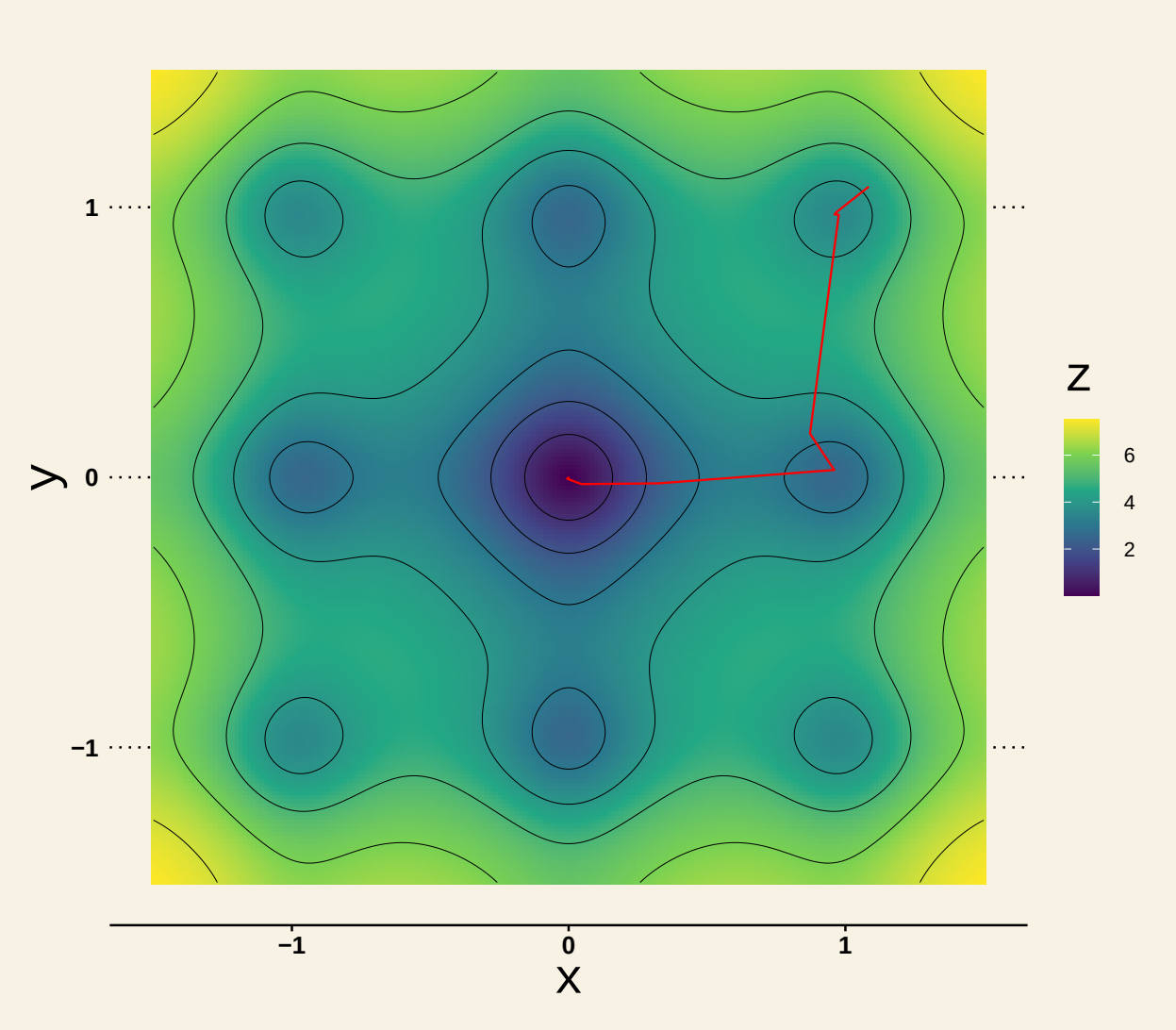
```
## seed =  0 , mean fitness =  0.6612734
## seed =  1 , mean fitness =  0.6509792
## seed =  2 , mean fitness =  0.6632172
## seed =  3 , mean fitness =  0.6325777
## seed =  4 , mean fitness =  0.6968631
## seed =  5 , mean fitness =  0.6336284
## seed =  6 , mean fitness =  0.6556191
## seed =  7 , mean fitness =  0.6442573
## seed =  8 , mean fitness =  0.6555563
## seed =  9 , mean fitness =  0.63381
```

Również dla tej funkcji algorytm osiągnął zadowalające wyniki.

Następnie zwizualizowana zostanie droga najlepszego osobnika.

```
set.seed(0)
path_history <- optimize.evolution(
    fitness_function = fitness_func,
    pop_size = 2000,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 0.22,
    elite_size = 1500,
    tournament_size = 3,
    budget = 1e+5,
    maximize = F,
    save_path = T)$path_history

prepare.plot2D(fitness_func, -1.5, 1.5, -1.5, 1.5, 200) +
    annotate("function", x = mapply(function(x) x[[1]], path_history),
             y = mapply(function(x) x[[2]], path_history), color = "red", size = 0.5)
```



Również dla tej funkcji najlepszy osobnik bardzo szybko znalazł się w okolicach optimum.

Teraz można przejść do badania wlpwu parametru k sukcesji elitarnej na średnie wyniki.

```
examine.elite_size(function(k) optimize.evolution(
    fitness_function = fitness_func,
    pop_size = 2000,
    starting_distribution = starting_dist,
    mutation_chance = 1,
    mutation_strength = 0.22,
    elite_size = k,
    tournament_size = 3,
    budget = 1e+5,
    maximize = F,
    save_path = T),
    elite_size_range = seq(1, 2000, 100))
```

```
## elite_size =  1 , mean fitness =  2.527601
## elite_size =  101 , mean fitness =  1.765142
## elite_size =  201 , mean fitness =  1.693764
```

```
## elite_size =  301 , mean fitness =  1.612899
## elite_size =  401 , mean fitness =  1.521798
## elite_size =  501 , mean fitness =  1.439256
## elite_size =  601 , mean fitness =  1.34922
## elite_size =  701 , mean fitness =  1.229291
## elite_size =  801 , mean fitness =  1.137081
## elite_size =  901 , mean fitness =  1.068463
## elite_size =  1001 , mean fitness =  1.015992
## elite_size =  1101 , mean fitness =  0.9188229
## elite_size =  1201 , mean fitness =  0.8486194
## elite_size =  1301 , mean fitness =  0.7827014
## elite_size =  1401 , mean fitness =  0.7310543
## elite_size =  1501 , mean fitness =  0.6443838
## elite_size =  1601 , mean fitness =  0.645242
## elite_size =  1701 , mean fitness =  0.7051624
## elite_size =  1801 , mean fitness =  0.7593343
## elite_size =  1901 , mean fitness =  1.07041
```

Dla funkcji $f_2(x)$ zwiększanie parametru `elite_size` powoduje polepszenie wyniku, jednak po przekroczeniu progu około `elite_size = 1500` średnie dopoasowanie zaczyna się pogarszać. Jako że funkcja $f_2(x)$ ma więcej ekstremów lokalnych niż funckja $f_1(x)$, to po przekroczeniu pewnej wartości eksploracja przestaje być wystarczająca i algorytm ma większe problemy z wydostawaniem się z optimów lokalnych. Przez to zjawisko, pomimo szybszej zbieżności, większość populacji osiąga okolice ekstremum globalnego za późno, by zdążyć skorzystać z tej własności i osiągnąć zadowalający wynik.