Dokumentacja wstępna

ALHE 2018 – algorytm mcga

Machine Coded Genetic Algorithm

Opis:

Algorytm mutacyjny do rozwiązywania problemów optymalizacyjnych o postaci liczby zmiennoprzecinkowej. Algorytm używa liczb o reprezentacji bajtowej, nie zmiennoprzecinkowej. Każdy osobnik jest wektorem bajtowym.

Algorytm polega na klasycznym, inspirowanym biologią, krzyżowaniu się chromosomów – rodziców, przez co powstaje lista dwójki ich potomstwa oraz na mutacji danych osobników. Mutacja jest typu *byte_mutation*, zaś krzyżowanie jest typu *byte_crossover*.

Mcga2 jest rozszerzoną wersją mcga, pozwalającą na większą kontrolę nad algorytmem.

Lista parametrów mcga:

- ***popsize*** – wielkość populacji (liczba chromosomów)
- ***chsize*** – wielkość wektora bajtów (chromosomu)
- ***crossprob*** – prawdopodobieństwo krzyżowania (domyślnie 1.0)
- ***mutateprob*** – prawdopodobieństwo mutacji (domyślnie 0.01)
- ***elitism –*** liczba najlepszych osobników, które będą przekopiowane do następnej populacji (domyślnie 1)
- ***minval*** – minimalna wartość dla wektora bajtów zamienionego na liczbę rzeczywistą dla populacji początkowej
- ***maxval*** - maksymalna wartość dla wektora bajtów zamienionego na liczbę rzeczywistą dla populacji początkowej
- ***maxiter*** – maksymalna liczba pokoleń
- ***evalFunc –*** funkcja na podstawie, której obliczane jest dopasowanie dla danego osobnika

Lista parametrów mcga2:

- ***crossover*** – metoda użyta do krzyżowania osobników
- ***fitness*** - funkcja na podstawie, której obliczane jest dopasowanie dla danego osobnika
- **…** - argumenty dodatkowe dla funkcji obliczającej dopasowanie
- ***min*** – wektor minimalnych wartości dla zmiennych
- ***max*** – wektor maksymalnych wartości dla zmiennych
- ***population*** – populacja początkowa
- ***selection*** – operator selekcji
- ***crossover*** – typ krzyżowania stosowanego w algorytmie (domyślnie *byte_crossover)*
- ***mutation*** – typ mutacji stosowanej w algorytmie (domyślnie *byte_mutation)*
- ***popSize*** – wielkość populacji (domyślnie 50)
- ***pcrossover*** – prawdopodobieństwo krzyżowania (domyślnie 0.8)
- ***pmutation*** – prawdopodobieństwo mutacji (domyślnie 0.1)
- ***elitism*** – liczba najlepszych osobników, które będą przekopiowane do następnej populacji (domyślnie 1)
- ***maxiter*** – maksymalna liczba pokoleń
- run – po ilu generacjach algorytm ma się zatrzymać jeśli nie ma lepszego rozwiązania (domyślnie *maxiter*)
- ***maxFitenss*** – maksymalna wartość dopasowania (domyślnie *infinite*)
- ***names*** – wektor nazw zmiennych
- ***paralel*** – dopasowanie dla osobników obliczane równolegle (domyślnie *false*)
- ***monitor*** – funkcja do wyświetlania danego stanu populacji
- ***seed*** – zalążek dla generatora liczb losowych

Rodzaje krzyżowań:

arithmetic_crossover (krzyżowanie arytmetyczne):

losowana jest wartość wagi – współczynnik *alfa* z rozkładu jednostajnego (od 0 do 1), po czym potomstwo powstaje poprzez przemnożenie współczynnika *alfa* poprzez wszystkie bajty dla jednego rodzica zsumowane z przemnożonymi bajtami drugiego rodzica przez wagę (1 – *alfa),* drugi potomek powstaje tak samo, tylko z rodzice są w odwrotnej kolejności przy mnożeniu (pierwszy rodzic mnożony przez (1 – *alfa*), drugi przez *alfa*)

blx_crossover:

blendowanie osobników, dla kolejnych bajtów w wektorze rodziców, losowane są wartości spomiędzy bajtu rodzica pierwszego i drugiego z pewnymi odchyleniami i tworzone są dwa osobniki potomne

byte_crossover (bajtowe krzyżowanie):

dla każdej pary bajtów w wektorach rodziców losowana jest wartość z rozkładu jednostajnego przyjmująca wartość 1 - bajty w wektorze rodziców zamieniają się miejscami, bądź 0 - bajty pozostają na swoich miejscach

byte_crossover_1p (bajtowe krzyżowanie jednopunktowe):

losowany jest punkt z przedziału [0, liczba bajtów w wektorze], który staje się punktem 'cięcia', po czym zamieniane są pomiędzy rodzicami bajty aż do wylosowanego indeksu

byte_crossover_2p (bajtowe krzyżowanie dwupunktowe):

działa tak samo jak *byte_crossover_1p,* jednak 'tnie' wektor w dwóch punktach (losuje dwie wartości z przedziału) i zamienia tylko środkową część wektorów

flat_crossover (krzyżowanie płaskie):

dla każdego bajtu w wektorze losuje liczbę z zakresu min-max dla danego bajtu

linear_crossover (krzyżowanie liniowe):

generowani są trzej potomkowie, na podstawie wzorów:

1) 0.5 * p1 + 0.5 * p2
2) 1.5 * p1 – 0.5 * p2
3) -0.5 * p1 + 1.5 * p2

gdzie p1 – rodzic pierwszy, p2 – rodzic drugi, po czym następuje selekcja, w której wybierani są dwaj najlepsi potomkowie

sbx_crossover (symulowane krzyżowanie binarne):

bajty dla potomków są wyliczane na jako średnia z obu rodziców +/- połowa różnicy między rodzicami, pomnożona przez współczynnik *beta* (wzór na współczynnik *beta* dla każdego bajtu jest losowany spomiędzy dwóch różnych, z występującym parametrem *nc* równym 50)

unfair_average_crossover (niesprawiedliwe średnie krzyżowanie):

losowany jest współczynnik *alfa* z rozkładu normalnego (od 0 do 0.5) po czym losowany jest indeks *j,* który mówi do którego miejsca w wektorze tworzonych potomków, używać danych wzorów:

1) Jeśli $i <= j$:
   $off1 = (1 + alfa) * p1_i - alfa * p2_i$
   $off2 = - alfa * p1_i + (1 + alfa) * p2_i$
2) Jeśli $i > j$
   $off1 = (1 - alfa) * p1_i + alfa * p2_i$
   $off2 = alfa * p1_i + (1 - alfa) * p2_i$

Rodzaje mutacji:

byte_mutation (mutacja bajtowa):

polega na zmianie losowo wybranego bajtu w osobniku dodając bądź odejmując od niego 1 (+1 z prawdopodobieństwem ½ i -1 z prawdopodobieństwem ½)

byte_mutation_dynamic (dynamiczna mutacja bajtowa):

zachowuje się identycznie jak *byte_mutation*, jedynie czynnik prawdopodobieństwa mutacji wraz z kolejnymi pokoleniami maleje

byte_mutation_random (losowa mutacja bajtowa):

dla losowo wybranego bajtu w osobniku, zmienia jego wartość na wylosowaną z przedziału [0,255]

byte_mutation_random_dynamic (losowa mutacja bajtowa):

zachowuje się identycznie jak *byte_mutation_random*, jedynie czynnik prawdopodobieństwa mutacji wraz z kolejnymi pokoleniami maleje

Przykładowe uruchomienie metody dla optymalizacji funkcji kwadratowej:

```
# A sample optimization problem
# Min f(xi) = (x1-7)^2 + (x2-77)^2 + (x3-777)^2 + (x4-7777)^2 + (x5-77777)^2
# The range of xi is unknown. The solution is
# x1 = 7
# x2 = 77
# x3 = 777
# x4 = 7777
# x5 = 77777
# Min f(xi) = 0
require("mcga")
 f<-function(x){
    return ((x[1]-7)^2 + (x[2]-77)^2 +(x[3]-777)^2 +(x[4]-7777)^2 +(x[5]-77777)^2)
 }
 m <- mcga(  popsize=200,
                   chsize=5,
                   minval=0.0,
                   maxval=999999999.9,
                   maxiter=2500,
                   crossprob=1.0,
                   mutateprob=0.01,
                   evalFunc=f)

 cat("Best chromosome:\n")
 print(m$population[1,])
 cat("Cost: ",m$costs[1],"\n")
```