Documentatie Laborator 4

Constantinescu Ana-Gabriela, 331

**Algoritm evloutiv**

Structura algoritmului meu evolutiv

* variabila *all\_best\_fit* a fost folosita pentru a putea reprezenta un grafic cu evolutia besturilor, pe parcursul tuturor generatiilor.

def genetic\_alg(*n*, *population\_size*, *parents\_size*, *generation\_number*)  
 *population* = generate\_random\_population(*population\_size*, *n*)  
 *gen* = 0  
 *all\_best\_fit* = []  
 *all\_best\_fit* << *population*.min\_by { |*p*| *p*.fitness }.fitness  
 *generation\_number*.times do  
 *gen* += 1  
 *parents* = generate\_parents(*parents\_size*, *population*)  
 *children* = crossover(*parents*, *n*)  
 *mutated\_children* = mutate\_children(*children*, *n*)  
 *population* = select\_survivors(*children*, *mutated\_children*, *population\_size*)  
 *all\_best\_fit* << *population*.first.fitness  
 end  
 *all\_best\_fit*end

Clasa Chromosome - in care voi retine pozitia fiecarui cromozom si fitnessul. Fitnessul este calculate in interiorul clasei, acesta nu trebuie dat ca parametru:

class *Chromosome* attr\_reader :solution, :fitness  
  
 def initialize(*solution*)  
 @solution = *solution* @fitness = calculate\_fitness  
 end  
  
 private  
  
 def calculate\_fitness  
 *n* = solution.length  
 *fit* = 0  
 (0..*n* - 1).each do |*i*|  
 *fit* += solution[*i*] \*\* 2  
 end  
 *fit* end  
end

**Pasii unui algoritm evolutiv**

Generarea populatiei initiale

def generate\_random\_x(*n*)  
 *x* = []  
 (0..*n* - 1).each do |*i*|  
 *x* << rand(-5.12..5.12)  
 end  
 *x*end  
  
def generate\_random\_population(*population\_size*, *n*)  
 *population* = []  
 (1..*population\_size*).each do  
 *population* << Chromosome.new(generate\_random\_x(*n*))  
 end  
 *population*end

Generarea parintilor – algoritm de tip Turnir cu k = 3

def generate\_parents(*parents\_size*, *population*)  
 *parents* = []  
 *pop\_size* = *population*.length  
 (1..*parents\_size*).each do  
 *best\_fit* = 9999999  
 *c* = nil  
 3.times do  
 *curr\_c* = *population*[rand(*pop\_size* - 1)]  
 *curr\_fit* = *curr\_c*.fitness  
 if *curr\_fit* < *best\_fit  
 best\_fit* = *curr\_fit  
 c* = *curr\_c* end  
 end  
 *parents* << *c* end  
 *parents*end

Incrucisare – de tip medie completa

Din 2 parinti, se creeaza un singur copil. Astfel, eu am generat de 2 ori mai multi parinti decat populatie initiala, pentru a rezulta un numar de copii egal cu populatia initiala

def crossover(*parents*, *n*)  
 *children* = []  
 *i* = 0  
 (*parents*.length / 2).times do  
 *p1* = *parents*[*i*]  
 *p2* = *parents*[*i* + 1]  
 *c* = get\_child(*p1*.solution, *p2*.solution, *n*)  
 *children* << Chromosome.new(*c*)  
 *i* += 2  
 end  
 *children*end  
  
*# incrucisare medie completa*def get\_child(*p1*, *p2*, *n*)  
 *c* = []  
 (0..*n* - 1).each do |*i*|  
 *c* << (*p1*[*i*] + *p2*[*i*]) / 2  
 end  
 *c*end

Mutatie – schimbarea unei singure gene

def mutate\_children(*children*, *n*)  
 *mutated\_children* = []  
 *children*.each do |*child*|  
 *curr\_sol* = *child*.solution  
 *rand\_pos* = rand(*n* - 1)  
 *new\_gene* = rand(-5.12..5.12)  
 *mutation* = []  
 (0..*n* - 1).each do |*i*|  
 if *i* == *rand\_pos  
 mutation* << *new\_gene* else  
 *mutation* << *curr\_sol*[*i*]  
 end  
 end  
 *mutated\_children* << Chromosome.new(*mutation*)  
 end  
 *mutated\_children*end

Selectia supravietuitorilor – de tip (miu, lambda)

def select\_survivors(*children*, *mutated\_children*, *population\_size*)  
 *survivors* = []  
 *children*.each { |*p*| *survivors* << *p*}  
 *mutated\_children*.each { |*p*| *survivors* << *p*}  
 *survivors*.sort\_by! { |*s*| *s*.fitness }  
 *survivors*.first(*population\_size*)  
end

Metoda main – are ca scop rularea de 10 a algoritmului evolutive si scrierea in fisiere a rezultatelor

def main  
 *all\_data* = []  
 *only\_best\_fit* = []  
 10.times do  
 *t0* = Time.now  
 *best\_fit* = genetic\_alg(50, 200, 400, 1000)  
 *all\_data* << *best\_fit*.to\_s  
 *only\_best\_fit* << *best\_fit*.last  
 *all\_data* << (Time.now - *t0*).to\_s  
 end  
 *all\_data* << "Best fit: #{*only\_best\_fit*.min}"  
 *all\_data* << "Worst fit: #{*only\_best\_fit*.max}"  
 *all\_data* << "Average fit: #{*only\_best\_fit*.sum/10}"  
 *write\_data* = *all\_data*.join("\n")  
 File.open("pop\_200\_gen\_1000\_n\_50", "w") { |*f*| *f*.write *write\_data* }  
end

**Rezultate**

Minimul functiei mele avea valoarea 0, pentru punctul xi = 0, ∀ ar fi i din {1, 2, …, n}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | Nr populatie | Nr generatii | Best | Worst | Average |
| 5 | 50 | 100 | 6.027e-06 | 0.021 | 0.004 |
| 5 | 100 | 200 | 1.556e-06 | 0.002 | 0.0002 |
| 5 | 200 | 400 | 1.812e-09 | 0.001 | 0.0002 |
| 20 | 50 | 100 | 0.083 | 1.206 | 0.415 |
| 20 | 100 | 200 | 0.01 | 0.25 | 0.069 |
| 20 | 200 | 400 | 0.003 | 0.402 | 0.059 |
| 20 | 200 | 800 | 0.0004 | 0.065 | 0.02 |
| 20 | 200 | 1000 | 6.773e-05 | 0.06 | 0.019 |
| 50 | 50 | 100 | 1.397 | 2.905 | 2.289 |
| 50 | 100 | 200 | 0.245 | 0.986 | 0.486 |
| 50 | 200 | 400 | 0.034 | 0.313 | 0.101 |
| 50 | 200 | 800 | 0.011 | 0.465 | 0.137 |
| 50 | 200 | 1000 | 0.004 | 0.222 | 0.037 |

**Swarm Intelligence**

Clasa Particle – creata pentru a retina pentru fiecare particular:

* pozitia sa
* fitnessul
* viteza
* cea mai buna varianta a particulei respective

class *Particle* attr\_accessor :position, :fitness, :speed, :best  
  
 def initialize(*position*, *speed*)  
 @position = *position* @speed = *speed* @fitness = calculate\_fitness(*position*)  
 @best = *position* end  
  
 def calculate\_fitness(*position*)  
 *n* = *position*.length  
 *fit* = 0  
 (0..*n* - 1).each do |*i*|  
 *fit* += *position*[*i*] \*\* 2  
 end  
 *fit* end  
  
 def update\_fitness  
 @fitness = calculate\_fitness(position)  
 end  
end

Generarea populatiei initiale

def generate\_initial\_speed(*n*)  
 *speed* = []  
 *n*.times do  
 *speed* << rand(-1.0..1.0)  
 end  
 *speed*end  
  
def generate\_random\_position(*n*)  
 *x* = []  
 *n*.times do  
 *x* << rand(-5.12..5.12)  
 end  
 *x*end  
  
def generate\_population(*population\_size*, *n*)  
 *population* = []  
 *population\_size*.times do  
 *population* << Particle.new(generate\_random\_position(*n*), generate\_initial\_speed(*n*))  
 end  
 *population*end

Algoritmul general

def pso(*population\_size*, *n*, *generation\_number*, *w*, *c1*, *c2*, *v\_max*)  
 *population* = generate\_population(*population\_size*, *n*)  
 *generation* = 0  
 *generation\_number*.times do  
 *generation* += 1  
 *best\_fit* = *population*.first.fitness  
 *global\_best* = nil  
 *population*.each do |*particle*|  
 *curr\_fit* = *particle*.fitness  
 if *curr\_fit* < *best\_fit  
 best\_fit* = *curr\_fit  
 global\_best* = *particle*.position  
 end  
 end  
 p calculate\_fitness(*global\_best*)  
  
 *population*.each do |*particle*|  
 *pos* = []  
 *speed* = []  
 *p\_best* = *particle*.best  
 *p\_speed* = *particle*.speed  
 *p\_pos* = *particle*.position  
 (0..*n* - 1).each do |*i*|  
 *speed*[*i*] = *w* \* *p\_speed*[*i*] + *c1* \* rand \* (*p\_best*[*i*] - *p\_pos*[*i*]) + *c2* \* rand \* (*global\_best*[*i*] - *p\_pos*[*i*])  
 if *speed*[*i*] > *v\_max  
 speed*[*i*] = *v\_max* elsif *speed*[*i*] < -*v\_max  
 speed*[*i*] = -*v\_max* end  
 *pos*[*i*] = *p\_pos*[*i*] + *speed*[*i*]  
 if *pos*[*i*] > 5.12  
 *pos*[*i*] = 5.12  
 elsif *pos*[*i*] < -5.12  
 *pos*[*i*] = -5.12  
 end  
 end  
 *particle*.speed = *speed  
 previous\_fit\_for\_p\_best* = calculate\_fitness(*particle*.best)  
 *particle*.position = *pos  
 particle*.update\_fitness  
 if *previous\_fit\_for\_p\_best* > *particle*.fitness  
 *particle*.best = *pos* end  
 end  
 end  
end