Computação Evolucionária

2013/2014

Notes for the PL 4
Version II

Ernesto J. F. Costa

8 de Março de 2014

Class PL 4

Evolutionary Algorithms

4.1 Introduction

We discussed in class the structure of an evolutionary algorithm, that can be easily defined in pseudocode (see algorithm 1).

Algoritmo 1: Simple Evolutionary Algorithm

```
  Input: NumGener, Problem
  Output: Best
1 Population ← RandomPopulation(Problem);
2 Population ← EvalPopulation(Population);
3 foreach $gener_i \in$ NumGener do
4     Mates ← SelectParents(Problem);
5     Offspring ← Variation(Mates);
6     Offspring ← EvalPopulation(Population);
7     Population ← SelectSurvivors(Population, Offspring);
8 return Best(Population);
```

The main issues when applying this model to a concrete problem are **how to**:

- Choose a **representation** for the candidate solutions
- Define a **fitness** function
- Choose the **variation** operators
- Choose the **selection** mechanisms

To exemplify let’s consider the case already discussed of the **Numbers of João Brandão**. Without many thought, the decisions are the following.

We will be using a binary representation; the fitness function will maximize the length of the solution, while minimizing the number of violations of the constraint that imposes that no number in the solution can be the average of two others also present in that same solution; we will choose a binary flip mutation and a one-point crossover as variation operators; the parents will be selected by tournament and the selection of the survivors will be generational with elitism. This is what is implemented in the code below. Nevertheless, keep in mind that because these options are arguments of the main function (`sea`) it a general template that you can costumize by I’m,plementing different versions of the mechanisms and of the variation operators. Now the code.

```
from random import random,randint, shuffle, uniform,sample
from operator import itemgetter
from math import sqrt
from copy import deepcopy

def sea(numb_generations,size_pop, size_cromo, prob_mut, prob_cross,selection,recombination,mutation,survivors, fit_func, phenotype,elite,t_size):
    if int(size_pop * elite) == 0:
        print('No Elitism will be used!!!')
    #  initialize population: [...,indiv,....] with indiv = (cromo,fit)
    populacao = [(gera_indiv(size_cromo),0) for j in range(size_pop)]
    # evaluate population
    populacao = [[indiv[0], fit_func(indiv[0])] for indiv in populacao]
    populacao.sort(key=itemgetter(1), reverse = True) # Maximizing!
    for i in range(numb_generations):
  # work with a copy of the population
        aux_populacao = deepcopy(populacao)
        # select parents
        mate_pool = selection(aux_populacao, t_size)
  # Variation
  # ------ Crossover
```

```
        progenitores = []
        for i in  range(0,size_pop-1,2):
            cromo_1= mate_pool[i][0]
            cromo_2 = mate_pool[i+1][0]
            filhos = recombination(cromo_1,cromo_2, prob_cross)
            progenitores.extend(filhos)
        # ------ Mutation
        descendentes = []
        for cromo,fit in progenitores:
            novo_cromo = cromo
            novo_cromo = mutation(cromo,prob_mut)
            descendentes.append((novo_cromo,0))
        descendentes = [ [indiv[0], fit_func(indiv[0])] for indiv in descendentes]
        descendentes.sort(key=itemgetter(1), reverse = True)
        # New population
        populacao = survivors(populacao,descendentes,elite)
        # Evaluate and sort
        populacao = [[indiv[0], fit_func(indiv[0])] for indiv in populacao]
        populacao.sort(key=itemgetter(1), reverse = True) # Maximizing
    print("Individual: %s\nSize: %s\nFitness: %4.2f\nViolations:%d" % (phenotype(populacao[0][0]), len(phenotype(populacao[0][0])), populacao[0][1],viola(phenotype(populacao[0][0]), size_cromo)))

    return 0
```

Listagem 4.1: General code

As you can see the code follows very closely the pseudocode. Be aware that each individual has two components: the chromosome and its fitness. It it represented by a python tuple. Notice also that the variation operators are applied to the chromosome, not to the whole individual. Finally, after the evaluation of the populations we order them, for that make some of the tasks easier.

Now the implementation related with the problem[1].

[1]Here we also tried to keep the code as general as possible, so it can be used with some other problems involving binary representations.

```python
# Representation by binary strings
def gera_indiv(size_cromo):
    indiv = [randint(0,1) for i in range(size_cromo)]
    return indiv

# mutation: chromosome
def muta_bin(indiv,prob_muta):
    cromo = indiv[:]
    for i in range(len(indiv)):
        cromo[i] = muta_bin_gene(cromo[i],prob_muta)
    return cromo

#  mutation: gene
def muta_bin_gene(gene, prob_muta):
    g = gene
    value = random()
    if value < prob_muta:
        g ^= 1
    return g

# Crossover
def one_point_cross(cromo_1, cromo_2,prob_cross):
  value = random()
  if value < prob_cross:
    pos = randint(0,len(cromo_1))
    f1 = cromo_1[0:pos] + cromo_2[pos:]
    f2 = cromo_2[0:pos] + cromo_1[pos:]
    return [f1,f2]
  else:
    return [cromo_1,cromo_2]

# Tournament Selection
def tournament_selection(population,t_size):
    size= len(population)
    mate_pool = []
    for i in range(size):
        winner = tournament(population,t_size)
        mate_pool.append(winner)
    return mate_pool

def tournament(population,size):
    """Maximization Problem.Deterministic"""
    pool = sample(population, size)
    pool.sort(key=itemgetter(1), reverse=True)
    return pool[0]

# Survivals: elitism
def survivors_elitism(parents,offspring,elite):
    """ Assunption: no size problems"""
    size = len(parents)
    comp_elite = int(size* elite)
    new_population = parents[:comp_elite] + offspring[:size -
       comp_elite]
    return new_population

# Fitness: João Brandão
def merito(indiv):
    return evaluate(phenotype(indiv), len(indiv))

def phenotype(indiv):
    fen = [ i+1 for i in range(len(indiv)) if indiv[i] == 1]
    return fen

def evaluate(indiv, comp):
    alfa = 1.5
    beta = -1
    return alfa * len(indiv) + beta * viola(indiv,comp)

def viola(indiv,comp):
    # Count violations
    v=0
    for elem in indiv:
      limite= min(elem-1,comp - elem)
      vi=0
      for j in range(1,limite+1):
        if ((elem - j) in indiv) and ((elem+j) in indiv):
          vi+=1
      v+=vi
    return v
```

Listagem 4.2: João Brandão's Numbers

Now it is time for you analyze carefully the code and understand how it works. This code can be downloaded from **InforEstudante** (the file name is `jb_sea.py`).

4.2 Warmup

We will start by extending the work done at the theoretical class concerning the João Brandão's Numbers problem.

Problema 4.1 **F**

Complete the code so you can **plot** the evolution of the best element of the population over the generations and of the average fitness of the entire population, also over the generations. Analyze the results and draw conclusions.

Problema 4.2 **F**

Download the code for the simple evolutionary algorithm applied to the João Brandão's Numbers problem. Try to find out which combination of the several parameters gives the best results, i.e., maximizing the length while minimizing the number of violations. The experiments that you need to do must be done in a controlled way. That means that you should define a table (see table 4.1) of possible values for each possible parameter.

Parameter	Values
Problem Instance	{$\underline{20}$,50,100}
Population Size	{100,$\underline{250}$,500}
Number Generations	{50,$\underline{100}$,500}
Prob. Mutation	{0.01,0.05,$\underline{0.1}$,0.3}
Prob. Crossover	{0.3, 0.6,$\underline{0.9}$}
Elite Size	{0.02, $\underline{0.05}$, 0.1}
Tournament Size	{$\underline{3}$,5,7}

Tabela 4.1: Values for the parameters

As you can see the number of tests that one have to do is huge (and equal to 3888!!!) and thus very time consuming. It this impossible to do them in class. You should try this at home before the lab class, but be prepared to consider only a subset of the possible combinations The underlined values are the combination that you should try first [2].

Problema 4.3 **F**

Similar to the previous exercise. Write the code necessary to **run the evolutionary algorithm several times** with the same parametrization, keeping track **for each run** of the fitness of the best individual over the generations, and the average fitness of the population over the generations. After collecting the data you are going to manipulate them, by **merging the results of all runs**, defining a new set of data: the values of the best of the best over the generations, and the values of the average of the averages over the generations. Plot these results and draw your conclusions.

4.3 Benchmark Problems

We are going to review two problems, that we already discussed, i.e.: the 0/1 Knapsack Problem and the Rastrigin Function, A brief description of each problem follows. During the experiments you should use the best parameter that you obtain with the previous exercises. But remember that evolutionary solutions are tuned for particular problems, even though it may be possible that a particular parametrization is good enough for a certain **class** of problems.

4.3.1 0/1 Knapsack Problem

Description

The well-known single-objective 0/1 knapsack problem is defined as follows: a thief has a knapsack of a certain **capacity**. He looks around and sees different items of different **weights** and **values**. He would like the items he chooses for his knapsack to have the greatest value possible yet still fit in his knapsack. The problem is called the **0-1 Knapsack Problem** because the thief can either take (1) or leave (0) each item. The knapsack problem is an example of an integer linear problem which has NP-hard complexity.

The problem can be defined more formally as follows. Given a set of items (n, with $i = 1, ..., n$)), each with a weight w_i and a value v_i, the goal is

[2]If even that configuration takes to much time try to reduce the size of the problem.

to determine

$$max(\sum_{i=1}^{n} v_i \times x_i)$$

subject to:

$$\sum_{i=1}^{n} w_i \times x_i \leq C$$

with

$$x_i \in \{0, 1\}$$

.

Choosing the data

The hardness of the problem is sensitive to the choice of the data used (value, weight and maximum value). The weights and the value of each item are obtaining using uniform distributions in a certain interval. Basically, there are three ways to generate the data:

Uncorrelated

P[i] = uniform([1..v])
V[i] = uniform([1..v])

Weakly Correlated

P[i] = uniform([1..v])
V[i] = P[i] + uniform([-r..r])

Strongly Correlated

P[i] = uniform([1..v])
V[i] = P[i] + r

You should be aware that we do not allow negative values, i.e., you have to guarantee that $V[i] > 0$. Also, the more correlated the data the harder the problem.

Parâmetros

To fabricate the data we need to define concrete values for certain parameters. Typical values are:

- $v = 10$
- $r = 5$
- Número de itens $n \in \{100, 250, 500\}$

Finally, the capacity of the knapsack can be calculated in two different ways:

a) Restrictive: $C = 2 \times v$

b) Average: $C = 0.5 \times \sum_{i=1}^{n} P[i]$

4.3.2 Rastrigin Function

The Rastrigin function is an Highly multimodal with many local minima (see figure 4.1 for a 2D version). It is defined by the equation 4.1.

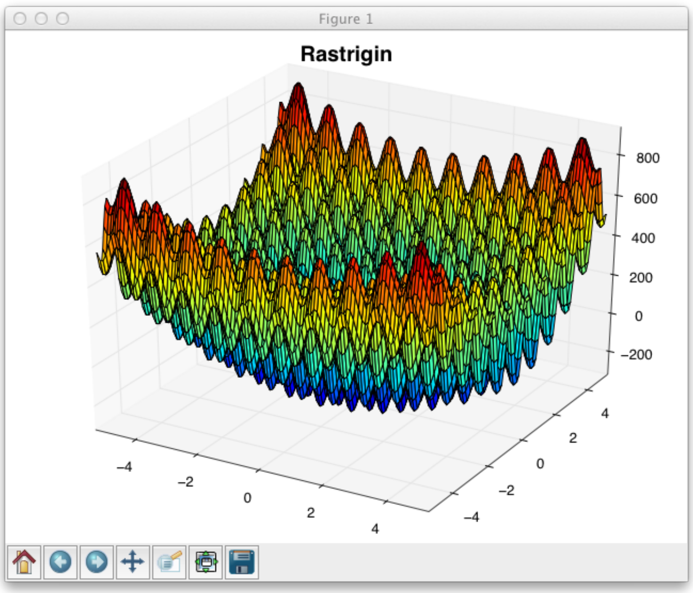

Figura 4.1: **Rastrigin**

$$f(\langle x_1, \ldots, x_n \rangle) = A \times n + (\sum_{i=1}^{n} x_i^2 - A \times cos(2\pi x_i) \qquad x_i \in [-5.12, 5.12] \quad (4.1)$$

with A typically equal to 10. The global minimum is zero, at point zero:

$$Min(Rastrigin) = Rastrigin(0, \ldots, 0) = 0$$

4.4 Exercises

This main goal of this exercise involves the implementation, test and comparison of the **hill-climbing**, the **parallel hill-climbing** and the **evolutionary algorithms** when applied to the two problems described above. For the representation issue you do not have to be concerned: use a binary representation for the 0/1 KP, and a vector of floats for the Rastrigin problem. Moreover, test the Rastrigin function for the case it has 10 dimensions.

Problema 4.4 F

Implement the parallel hill-climbing described in the theoretical lecture.

Problema 4.5 M

For each combination of the three algorithms and the two problems, **run 10 times** and collect the data of the best and average of the population. Besides looking for the figures do a visual analysis by plotting the results, presenting them divided by problem. Draw your conclusions.