**CPE-723 – Otimização Natural**

**Lista de Exercícios #4**

**Amanda Isabela de Campos (DRE 120074842)**

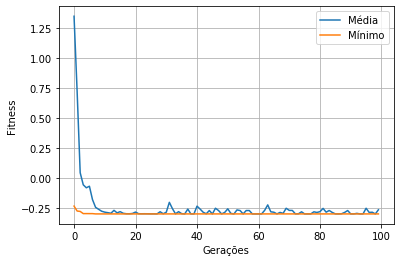
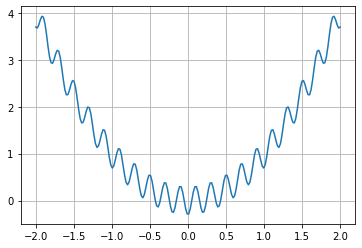
**UFRJ – COPPE – PEE / Programa de Engenharia El´etrica – CPE723 – Otimiza¸c˜ao Natural 2020/1 – Edi¸c˜ao Remota – Lista de Exerc´ıcios 4**

1. Escreva um algoritmo gen´etico simples (SGA) para minimiza¸c˜ao da fun¸c˜ao *y*(*x*) = *x*2 0*.*3 cos (10*πx*) no intervalo *x* [ 2*,* +2], utilizando um gen´otipo de representa¸c˜ao bin´aria com pelo menos 16 bits. Utilizando inicializa¸c˜ao aleat´oria, execute cinco vezes o algoritmo. Comente sobre os resultados obtidos.

∈ −

−

A Figura 1 (a) apresenta a função de minização do problema, onde observa-se que o ponto de mínimo (resultado esperado) é o ponto (0,-0.3). Na Figura 1 (b) está o resultado de execução do código de algoritmo genético simples (SGA) criado para minimizar a função, como forma de curva de progresso, onde observa-se que com 100 gerações o algoritmo converge para o valor esperado de aptidão de -0.3. O código criado para esse problema está reproduzido a seguir.



(a) (b)

Figura 1

import numpy as np

import matplotlib.pyplot as plt

import math

BITS = 16            #tamanho do cromossomo

POPULAÇÃO = 30           # tamanho da população

TAXA\_CROSS = 0.8         # probabilidade de crossover

TAXA\_MUTAÇÃO = 0.003   #probabilidade de mutação

N\_GENERAÇÕES = 50

X\_LIMITES = [-2,2]         #limites de x

def F(x): return -(x\*\*2-0.3\*np.cos(10\*x\*math.pi))    #máximo da função

# encontra as fitness diferentes de zero para seleção

def get\_fitness(pred): return pred + 1e-3 - np.min(pred)

# conversão binário para decimal normalizado no limite de X

def codificação(pop): return pop.dot(2 \*\* np.arange(BITS)[::-1]) / float(2\*\*BITS-1) \* X\_LIMITES[1]

def seleção(pop, fitness):    # SELEÇÃO

    idx = np.random.choice(np.arange(POPULAÇÃO), size=POPULAÇÃO, replace=True,

                           p=fitness/fitness.sum())

    return pop[idx]

def crossover(pai, pop):     #CROSSOVER

    if np.random.rand() < TAXA\_CROSS:

        i\_ = np.random.randint(0, POPULAÇÃO, size=1) #seleciona outro indivíduo da população

        cross\_points = np.random.randint(0, 2, size=BITS).astype(np.bool)   # pontos do crossover

        pai[cross\_points] = pop[i\_, cross\_points]                            # produz 1 filho

    return pai

def mutação(filho): # inversão de 1 bit

    for point in range(BITS):

        if np.random.rand() < TAXA\_MUTAÇÃO:

            filho[point] = 1 if filho[point] == 0 else 0

    return filho

pop = np.random.randint(2, size=(POPULAÇÃO, BITS))   #População inicial

i = 0

menor = np.zeros([N\_GENERAÇÕES])

media = np.zeros([N\_GENERAÇÕES])

for \_ in range(N\_GENERAÇÕES):

    F\_values = F(codificação(pop))

    fitness = get\_fitness(F\_values)

    pop = seleção(pop, fitness)

    pop\_copy = pop.copy()

    for pai in pop:

        filho = crossover(pai, pop\_copy)

        filho = mutação(filho)

        pai[:] = filho       # pai é substituído por seu filho

    menor[i] = -np.max(F\_values)

    media[i] = -np.mean(F\_values)

    i = i + 1

plt.plot(media, label='Média')

plt.plot(menor, label='Mínimo')

plt.legend()

plt.xlabel('Gerações')

plt.ylabel('Fitness')

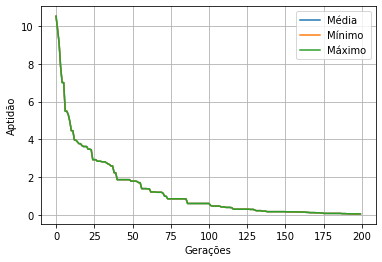
plt.grid()

print ('Média:', -np.mean(F\_values))

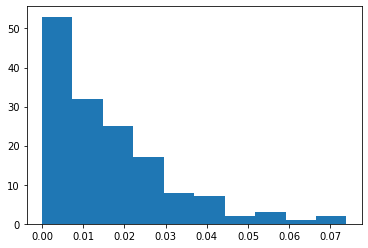
print ('Menor:', -np.max(F\_values))

1. **Escreva um algoritmo de estrat´egia de evolu¸c˜ao (ES) para encontrar o ponto ´otimo global da fun¸c˜ao de Ackley com 20 (ou mais) vari´aveis. Execute o algoritmo muitas vezes (150 execu¸c˜oes com inicializa¸c˜ao independente, por exemplo) e guarde o melhor resultado de cada execu¸c˜ao. No final, fa¸ca um histograma dos 150 valores armazenados.**

Um algoritmo de estratégia de evolução (ES) foi criado para encontrar o ponto de ótimo global da função Ackley com 20 variáveis. A curva de progresso para a primeira execução do algoritmo está apresenta a seguir, onde observa-se que o algoritmo com 200 gerações consegue convergir para o ponto de mínimo, como esperado.



A seguir o algoritmo foi executado 150 vezes com inicializações independentes e aleatórias, o histograma com os resultados está reproduzido a seguir, onde observa-se que a maior frequência de resultados está próxima de zero, como esperado. O código adotado neste problema está indicado na sequência.



# -\*- coding: utf-8 -\*-

"""

Created on Tue Sep  8 16:06:04 2020

@author: Amanda

"""

import numpy as np

import matplotlib.pyplot as plt

import math

N\_execuções = 150

hist = np.zeros([N\_execuções])

for ii in range(N\_execuções):

    DNA\_SIZE = 20            # DNA (real number)

    LIMITES = [-32.768, 32.768]        # limites superior e inferior da função

    N\_GERAÇÕES = 200

    TAM\_POP = 30           # tamanho da população

    N\_FILHOS = 200

    def F(x):

        firstSum = 0.0

        secondSum = 0.0

        for c in x:

            print(c)

            firstSum += c\*\*2

            secondSum += math.cos(2\*math.pi\*c)

        n = float(len(x))

        return -(-20.0\*math.exp(-0.2\*math.sqrt(firstSum/n)) - math.exp(secondSum/n) + 20 + math.e)

    def recombinação(pop, n\_filhos):

        filhos = {'DNA': np.empty((n\_filhos, DNA\_SIZE))}

        filhos['sigma'] = np.empty\_like(filhos['DNA'])

        for X\_filhos, Sigma\_filhos in zip(filhos['DNA'], filhos['sigma']):

            # Recombinação

            p1, p2 = np.random.choice(np.arange(TAM\_POP), size=2, replace=False)

            cp = np.random.randint(0, 2, DNA\_SIZE, dtype=np.bool)  # pontos de crossover

            X\_filhos[cp] = pop['DNA'][p1, cp]

            X\_filhos[~cp] = pop['DNA'][p2, ~cp]

            Sigma\_filhos[cp] = pop['sigma'][p1, cp]

            Sigma\_filhos[~cp] = pop['sigma'][p2, ~cp]

            # Mutação

            Sigma\_filhos[:] = np.exp(1/np.sqrt(20) \* np.random.randn(\*Sigma\_filhos.shape))\*Sigma\_filhos    # > 0

            X\_filhos += Sigma\_filhos \* np.random.randn(\*X\_filhos.shape)

            X\_filhos[:] = np.clip(X\_filhos, \*LIMITES)

        return filhos

    def seleção(pop, filhos):

        for key in ['DNA', 'sigma']:

            pop[key] = np.vstack((pop[key], filhos[key]))

        for i in range(230):

            fitness[i] = F(pop['DNA'][i,:])

        # fitness = custo\_fitness(F(pop['DNA']))

        ind = np.arange(pop['DNA'].shape[0])

        otim\_ind = ind[fitness.argsort()][-TAM\_POP:]

        for k in ['DNA', 'sigma']:

            pop[k] = pop[k][otim\_ind]

        return pop

    ## População inicial

    pop = dict(DNA=5 \* np.random.rand(1, DNA\_SIZE).repeat(TAM\_POP, axis=0),

               sigma=np.random.rand(TAM\_POP, DNA\_SIZE))

    # x = np.linspace(\*DNA\_BOUND, 200)

    # plt.plot(x, F(x))

    i = 0

    menor = np.zeros([N\_GERAÇÕES])

    media = np.zeros([N\_GERAÇÕES])

    maior = np.zeros([N\_GERAÇÕES])

    n = TAM\_POP + N\_FILHOS

    fitness = np.zeros([n])

    for i in range(N\_GERAÇÕES):

        filhos = recombinação(pop, N\_FILHOS)

        pop = seleção(pop, filhos)

        for k in range(TAM\_POP):

            menor[i] = -np.max(F(pop['DNA'][k,:]))

            maior[i] = -np.min(F(pop['DNA'][k,:]))

            media[i] = -np.mean(F(pop['DNA'][k,:]))

    # plt.plot(media, label='Média')

    # plt.plot(menor, label='Mínimo')

    # plt.plot(maior, label='Máximo')

    # plt.legend()

    # plt.xlabel('Gerações')

    # plt.ylabel('Aptidão')

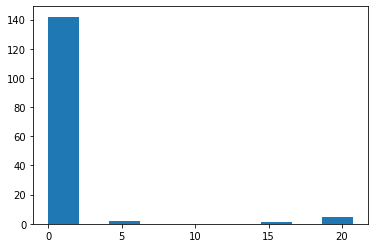
    # plt.grid()

    hist[ii] = min(maior)

plt.hist(hist)

1. **Repita o Exerc´ıcio 2, usando um algoritmo de programa¸c˜ao evolucion´aria (EP).**

O problema anterior foi solucionado com um algoritmo de programação evolucionária (EP), a seguir está indicado o histograma dos resultados de 150 execuções e o código criado. Observa-se que a maior parte das execuções convergiram para o ponto de mínimo igual a zero para todas as variáveis, porém em alguns casos o problema ficou preso em ótimos locais distantes do esperado, como 20 por exemplo, o que não aconteceu no problema anterior.



import numpy as np

import matplotlib.pyplot as plt

import math

N\_execuções = 150

hist = np.zeros([N\_execuções])

for ii in range(N\_execuções):

    DNA\_SIZE = 20            # DNA (real number)

    LIMITES = [-32.768, 32.768]        # limites superior e inferior da função

    N\_GERAÇÕES = 50

    TAM\_POP = 200           # tamanho da população

    def F(x):

        firstSum = 0.0

        secondSum = 0.0

        for c in x:

            print(c)

            firstSum += c\*\*2

            secondSum += math.cos(2\*math.pi\*c)

        n = float(len(x))

        return -(-20.0\*math.exp(-0.2\*math.sqrt(firstSum/n)) - math.exp(secondSum/n) + 20 + math.e)

    def mutação(pop):

        filhos = {'DNA': np.empty((TAM\_POP, DNA\_SIZE))}

        filhos['sigma'] = np.empty\_like(filhos['DNA'])

        for X\_filhos, Sigma\_filhos in zip(filhos['DNA'], filhos['sigma']):

            X\_filhos = pop['DNA']

            Sigma\_filhos = pop['sigma']

            X\_filhos = X\_filhos + Sigma\_filhos\*np.random.randn(\*X\_filhos.shape)

            Sigma\_filhos[:] = np.exp(1/np.sqrt(2\*20) \* np.random.randn(\*X\_filhos.shape))\*np.exp(1/np.sqrt(2\*np.sqrt(20)) \* np.random.randn(\*X\_filhos.shape))\*Sigma\_filhos

            X\_filhos[:] = np.clip(X\_filhos, \*LIMITES)

        return filhos

    def seleção(filhos):

        for key in ['DNA', 'sigma']:

            pop[key] = (filhos[key])

        return pop

    ## População inicial

    pop = dict(DNA=5 \* np.random.rand(1, DNA\_SIZE).repeat(TAM\_POP, axis=0),

              sigma=np.random.rand(TAM\_POP, DNA\_SIZE))

    # x = np.linspace(\*DNA\_BOUND, 200)

    # plt.plot(x, F(x))

    i = 0

    menor = np.zeros([N\_GERAÇÕES])

    media = np.zeros([N\_GERAÇÕES])

    maior = np.zeros([N\_GERAÇÕES])

    for i in range(N\_GERAÇÕES):

        filhos = mutação(pop)

        pop = seleção(filhos)

        for k in range(TAM\_POP):

            menor[i] = -np.max(F(pop['DNA'][k,:]))

            maior[i] = -np.min(F(pop['DNA'][k,:]))

            media[i] = -np.mean(F(pop['DNA'][k,:]))

    plt.plot(media, label='Média')

    plt.plot(menor, label='Mínimo')

    plt.plot(maior, label='Máximo')

    plt.legend()

    plt.xlabel('Gerações')

    plt.ylabel('Aptidão')

    plt.grid()

    hist[ii] = min(maior)

plt.hist(hist)

1. **Escreva e execute um algoritmo gen´etico simples (SGA) para resolver um dos trˆes problemas a seguir:**
   * **Problema de *N* rainhas, em um tabuleiro de tamanho *N* por *N* ;**

O algoritmo a seguir resolve o problema de N rainhas, em um tabuleiro N por N. Este foi executado com N = 8 e obteve como resultado

Aptidão máxima = 28

Solucionado na geração 4641!

Uma das soluções:

Cromossomo = [3, 5, 2, 8, 1, 7, 4, 6], Aptidão = 28

Forma do tabuleiro:

x x x Q x x x x

x x x x x Q x x

x x x x x x x Q

x Q x x x x x x

x x x x x x Q x

Q x x x x x x x

x x Q x x x x x

x x x x Q x x x

import random

def x(tamanho): #População inicial

    return [ random.randint(1, N\_rainhas) for \_ in range(N\_rainhas) ]

def fitness(cromossomo):

    colisoes\_horiz = sum([cromossomo.count(rainha)-1 for rainha in cromossomo])/2

    colisoes\_diag = 0

    n = len(cromossomo)

    diagonal\_esq = [0] \* 2\*n

    diagonal\_dir = [0] \* 2\*n

    for i in range(n):

        diagonal\_esq[i + cromossomo[i] - 1] += 1

        diagonal\_dir[len(cromossomo) - i + cromossomo[i] - 2] += 1

    colisoes\_diag = 0

    for i in range(2\*n-1):

        cont = 0

        if diagonal\_esq[i] > 1:

            cont += diagonal\_esq[i]-1

        if diagonal\_dir[i] > 1:

            cont += diagonal\_dir[i]-1

        colisoes\_diag += cont / (n-abs(i-n+1))

    return int(maxFitness - (colisoes\_horiz + colisoes\_diag)) #28-(2+3)=23

def prob(cromossomo, fitness):

    return fitness(cromossomo) / maxFitness

def seleção(população, probs):

    populaçãoWithProbabilty = zip(população, probs)

    total = sum(w for c, w in populaçãoWithProbabilty)

    r = random.uniform(0, total)

    upto = 0

    for c, w in zip(população, probs):

        if upto + w >= r:

            return c

        upto += w

    assert False, "Shouldn't get here"

def recombinação(x, y): #crossover de 2 cromossomos

    n = len(x)

    c = random.randint(0, n - 1)

    return x[0:c] + y[c:n]

def mutação(x):  #mutação de 1 valor aleatório

    n = len(x)

    c = random.randint(0, n - 1)

    m = random.randint(1, n)

    x[c] = m

    return x

def AG\_rainha(população, fitness):

    taxa\_mutação = 0.03

    nova\_população = []

    probs = [prob(n, fitness) for n in população]

    for i in range(len(população)):

        x = seleção(população, probs) #melhor cromossomo 1

        y = seleção(população, probs) #melhor cromossomo 2

        filhos = recombinação(x, y) #riação do filho

        if random.random() < taxa\_mutação:

            filhos = mutação(filhos)

        print\_cromossomo(filhos)

        nova\_população.append(filhos)

        if fitness(filhos) == maxFitness: break

    return nova\_população

def print\_cromossomo(crom):

    print("cromossomo = {},  Fitness = {}"

        .format(str(crom), fitness(crom)))

if \_\_name\_\_ == "\_\_main\_\_":

    N\_rainhas = int(input("N° de rainhas: "))

    maxFitness = (N\_rainhas\*(N\_rainhas-1))/2  # 8\*7/2 = 28

    população = [x(N\_rainhas) for \_ in range(100)]

    geração = 1

    while not maxFitness in [fitness(crom) for crom in população]:

        print("=== geração {} ===".format(geração))

        população = AG\_rainha(população, fitness)

        print("")

        print("Máxima Aptidão = {}".format(max([fitness(n) for n in população])))

        geração += 1

    crom\_saida = []

    print("Resolvido na geração {}!".format(geração-1))

    for crom in população:

        if fitness(crom) == maxFitness:

            print("");

            print("One of the solutions: ")

            crom\_saida = crom

            print\_cromossomo(crom)

    tab = []

    for x in range(N\_rainhas):

        tab.append(["x"] \* N\_rainhas)

    for i in range(N\_rainhas):

        tab[N\_rainhas-crom\_saida[i]][i]="Q"

    def tabuleiro(tab):

        for row in tab:

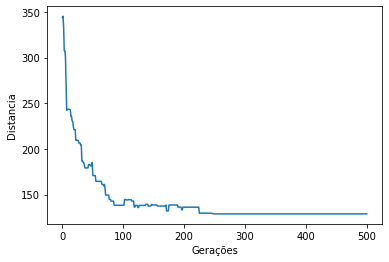
            print (" ".join(row))

    print()

    tabuleiro(tab)

* + **Problema do caixeiro viajante, com *K* cidades;**

O código a seguir resolve o problema do caixeiro viajante com 30 cidades, o resultado da curva de progresso está representado a seguir.



import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

class Cidades:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def distancia(self, city):

        xDis = abs(self.x - city.x)

        yDis = abs(self.y - city.y)

        distancia = np.sqrt((xDis \*\* 2) + (yDis \*\* 2))

        return distancia

    def \_\_repr\_\_(self):

        return "(" + str(self.x) + "," + str(self.y) + ")"

class Fitness:

    def \_\_init\_\_(self, rota):

        self.rota = rota

        self.distancia = 0

        self.fitness= 0.0

    def routeDistance(self):

        if self.distancia ==0:

            pathDistancia = 0

            for i in range(0, len(self.rota)):

                fromCity = self.rota[i]

                toCity = None

                if i + 1 < len(self.rota):

                    toCity = self.rota[i + 1]

                else:

                    toCity = self.rota[0]

                pathDistancia += fromCity.distancia(toCity)

            self.distancia = pathDistancia

        return self.distancia

    def routeFitness(self):

        if self.fitness == 0:

            self.fitness = 1 / float(self.routeDistance())

        return self.fitness

# Rota inicial aleatória

def CriaçãoRota(Listacidades):

    rota = random.sample(Listacidades, len(Listacidades))

    return rota

#População inicial

def PopulaçaoInicial(Tamanho\_população, Listacidades):

    população = []

    for i in range(0, Tamanho\_população):

        população.append(CriaçãoRota(Listacidades))

    return população

#Rank dos indivíduos

def rankRotas(população):

    aptidão = {}

    for i in range(0,len(população)):

        aptidão[i] = Fitness(população[i]).routeFitness()

    return sorted(aptidão.items(), key = operator.itemgetter(1), reverse = True)

#Seleção dos pais

def seleção(popRanked, taxaElit):

    selecionados = []

    df = pd.DataFrame(np.array(popRanked), columns=["Index","Fitness"])

    df['cum\_sum'] = df.Fitness.cumsum()

    df['cum\_perc'] = 100\*df.cum\_sum/df.Fitness.sum()

    for i in range(0, taxaElit):

        selecionados.append(popRanked[i][0])

    for i in range(0, len(popRanked) - taxaElit):

        pick = 100\*random.random()

        for i in range(0, len(popRanked)):

            if pick <= df.iat[i,3]:

                selecionados.append(popRanked[i][0])

                break

    return selecionados

#Recombinação

def matingPool(população, selecionados):

    matingpool = []

    for i in range(0, len(selecionados)):

        index = selecionados[i]

        matingpool.append(população[index])

    return matingpool

#Crossover: dois pais geram um filho

def Crossover(pai1, pai2):

    filho = []

    filho1 = []

    filho2 = []

    geneA = int(random.random() \* len(pai1))

    geneB = int(random.random() \* len(pai1))

    startGene = min(geneA, geneB)

    endGene = max(geneA, geneB)

    for i in range(startGene, endGene):

        filho1.append(pai1[i])

    filho2 = [item for item in pai2 if item not in filho1]

    filho = filho1 + filho2

    return filho

#Aplicação do crossover em toda a população

def CrossoverPopulação(matingpool, taxaElit):

    filhos = []

    length = len(matingpool) - taxaElit

    pool = random.sample(matingpool, len(matingpool))

    for i in range(0,taxaElit):

        filhos.append(matingpool[i])

    for i in range(0, length):

        filho = Crossover(pool[i], pool[len(matingpool)-i-1])

        filhos.append(filho)

    return filhos

#Mutação

def mutação(indivíduo, taxaMutação):

    for trocado in range(len(indivíduo)):

        if(random.random() < taxaMutação):

            trocar\_com = int(random.random() \* len(indivíduo))

            cidade1 = indivíduo[trocado]

            cidade2 = indivíduo[trocar\_com]

            indivíduo[trocado] = cidade2

            indivíduo[trocar\_com] = cidade1

    return indivíduo

#Aplicação da mutação em toda a população

def mutaçãoPop(população, taxaMutação):

    mutaçãoPop = []

    for ind in range(0, len(população)):

        mutatedInd = mutação(população[ind], taxaMutação)

        mutaçãoPop.append(mutatedInd)

    return mutaçãoPop

#Seleção dos sobreviventes

def prox\_Geração(Ger\_atual, taxaElit, taxaMutação):

    popRanked = rankRotas(Ger\_atual)

    Resultados\_selec = seleção(popRanked, taxaElit)

    matingpool = matingPool(Ger\_atual, Resultados\_selec)

    filho = CrossoverPopulação(matingpool, taxaElit)

    Prox\_ger = mutaçãoPop(filho, taxaMutação)

    return Prox\_ger

#Algoritmo genético principal

def geneticAlgorithm(população, Tamanho\_população, taxaElit, taxaMutação, gerações):

    pop = PopulaçaoInicial(Tamanho\_população, população)

    print("Initial distance: " + str(1 / rankRotas(pop)[0][1]))

    for i in range(0, gerações):

        pop = prox\_Geração(pop, taxaElit, taxaMutação)

    print("Final distance: " + str(1 / rankRotas(pop)[0][1]))

    melhorRotaIndex = rankRotas(pop)[0][0]

    melhorRota = pop[melhorRotaIndex]

    return melhorRota

#Criação da lista de cidades

Listacidades = []

N\_cidades = 30

for i in range(0,25):

    Listacidades.append(Cidades(x=int(random.random() \* N\_cidades), y=int(random.random() \* N\_cidades)))

#Execução do código

geneticAlgorithm(população=Listacidades, Tamanho\_população=30, taxaElit=20, taxaMutação=0.001, gerações=200)

#Gráfico de execução

def geneticAlgorithmPlot(população, Tamanho\_população, taxaElit, taxaMutação, gerações):

    pop = PopulaçaoInicial(Tamanho\_população, população)

    Progresso = []

    Progresso.append(1 / rankRotas(pop)[0][1])

    for i in range(0, gerações):

        pop = prox\_Geração(pop, taxaElit, taxaMutação)

        Progresso.append(1 / rankRotas(pop)[0][1])

    plt.plot(Progresso)

    plt.ylabel('Distancia')

    plt.xlabel('Gerações')

    plt.show()

geneticAlgorithmPlot(população=Listacidades, Tamanho\_população=100, taxaElit=20, taxaMutação=0.01, gerações=500)